1 a) $\log_2 n^2 + 1 \le cn$

For $c=2$ and $n_0 = 2$ $\Rightarrow \log_2 n^2 + 1 \le 2n \Rightarrow \underbrace{\log_2 2^2}_{2} + 1 \le 4 \Rightarrow 3 \le 4$ $\boxed{\text{True}}$

$n \ge 2$

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

$n\cdot(n+1) \ge c^2 n^2 \Rightarrow \frac{\cancel{n}\cdot(n+1)}{\cancel{n}} \ge \frac{c^2 n^{\cancel{2}}}{\cancel{n}} \Rightarrow n+1 \ge c^2 n \Rightarrow \underbrace{1 \ge n(c^2-1)} \Rightarrow \boxed{\text{False}}$

Impossible since $n(c^2-1)$ is bounded by a constant.

c) $\frac{c_1 n^n}{n^{n-1}} \le \frac{n^{n-1}}{n^{n-1}} \le \frac{c_2 n^n}{n^{n-1}}$

$\underbrace{c_1 n \le 1} \le c_2 n$

Impossible since $c_1 n$ is bounded by a constant. $\Rightarrow \boxed{\text{False}}$

2 $\lim\limits_{n\to\infty} \frac{n^{\cancel{2}}}{n^{\cancel{3}}_{n}} = 0 \Rightarrow n^2 = o(n^3) \Rightarrow \underline{n^2 < n^3}$

$\lim\limits_{n\to\infty} \frac{n^2}{\sqrt{n}} \stackrel{F'}{=} \frac{2n}{1/2\sqrt{n}} = 2n.2\sqrt{n} = 4n^{3/2} = \infty \Rightarrow \sqrt{n} = o(n^2) \Rightarrow \underline{\sqrt{n} < n^2}$

$\lim\limits_{n\to\infty} \frac{n^2\cancel{\log n}}{\cancel{\log n}} = 2n = \infty \Rightarrow \log n = o(n^2 \log n) \Rightarrow \underline{\log n < n^2 \log n}$

$\lim\limits_{n\to\infty} \frac{2^n}{10^n} = \left(\frac{\cancel{2}}{\cancel{10}_{5}}\right)^n = \frac{1}{5^n} = 0 \Rightarrow 2^n = o(10^n) \Rightarrow \underline{2^n < 10^n}$

$\lim\limits_{n\to\infty} \frac{2^n}{\underbrace{8^{\log_2 n}}_{n^{(\log_2 8)} 3}} = \frac{2^n}{n^3} \stackrel{F'}{=} \frac{2^n \ln 2}{3n^2} \stackrel{F''}{=} \frac{2^n . \ln^2(2)}{3n^2} \stackrel{F'''}{=} \frac{2^n . \ln^3(2)}{6} = \underbrace{\lim\limits_{n\to\infty} 2^n}_{\infty} . \underbrace{\frac{\ln^3(2)}{6}}_{\text{constant}} = \infty$

$8^{\log_2 n} = o(2^n) \Rightarrow \underline{8^{\log_2 n} < 2^n}$

Constant → Logorithmic → Linear → Log-linear
↓
Quadratic
↓
Cubic → Exponential → Factorial

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

(Growth rate -ascending order-)

a)

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

Inside of if : Θ(1)
Inside of else : Θ(1)
For loop : Θ(log(n))
Time complexity:
Θ(log(n))

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];
                        }
                }
        }
}
```

Let sizeofArray is n.
For loop : Θ(n)
Other statements : Θ(1)
Time complexity : Θ(n)

c)

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

There is only one simple statement.
Time complexity : Θ(1)

d)

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

There is a loop that is executed n/5 times and basic statement inside the loop.

Time complexity : Θ(1) + [Θ(n) * Θ(1)] + Θ(1) = 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]); -> Θ(1)

}
```

Outer loop : Θ(n)

Inner loop: Θ(log(n))

Time complexity: Θ(n) * Θ(log(n)) = Θ(n.log(n))

f)

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

Worst case : Θ(n) + Θ(n.log(n)) = Θ(n.log(n))

Best case : Θ(n) + Θ(n) = Θ(n)

General case : O(n.log(n))

g)

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

Print : Θ(1)

For loop : Θ(n)

While loop : Θ(log(n))

Time complexity : Θ(n.log(n))

h)

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

Print : Θ(1)

For loop : Θ(log(n))

While loop : Θ(log(n))

Time complexity : Θ(log^2(n))

i)

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

$T_B = \Theta(1) \Rightarrow$ Base case equality, multiply, subtraction ($==$, $*$, $-$)

$T_w \Rightarrow T_w(n) = T_w(n-1) + 3$, $(T(0) = 1)$

$= T_w(n-2) + 3.2$

$= T_w(n-3) + 3.3$

$\vdots$

$= T_w(\underbrace{n-k}_{0}) + 3.k$

$n-k = 0 \rightarrow k = n$

$T_w(n) = \underbrace{T(0)}_{1} + 3n = 3n+1 \Rightarrow T_w = \Theta(n)$

$\left.\begin{array}{l} T_B = \Theta(1) \\ T_w = \Theta(n) \end{array}\right\} \underline{\underline{O(n)}} \rightarrow$ General case

j)

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

$T_B = \Theta(1) \Rightarrow$ Base case ($==$, $-$)

$T_w \Rightarrow T_w(n) = T_w(n-1) + 2$, $(T(1) = 0)$

$= T_w(n-2) + 2.2$

$\vdots$

$= T_w(\underbrace{n-k}_{1}) + 2.k$

$n-k = 1 \rightarrow k = n-1$

$T_w(n) = \underbrace{T(1)}_{0} + 2(n-1) = 2n-2 \Rightarrow T_w = \Theta(n)$

$\left.\begin{array}{l} T_B = \Theta(1) \\ T_w = \Theta(n) \end{array}\right\} O(n) \rightarrow$ General case

$O(n) + \Theta(n) = \underline{\underline{O(n)}}$

④ a) An algorithm cannot take more time than the function ($f(N)$) of the big O notation ($T(N) \leq cF(N), \forall N \geq n_0$) that's why 'at least' part is meaningless for big O notation. 'At most' can be written instead of 'at least'.

b) I. $c_1 2^n \leq 2^{n+1} \leq c_2 2^n$

For $c_1 = 1, c_2 = 2, n_0 = 1 \Rightarrow \boxed{\text{True}}$
$(n \geq 1)$

$(2^n \leq 2^{n+1} \leq 2^{n+1})$

$n_0 = 1 \Rightarrow 2 \leq 2^2 \leq 2^2$

II. $c_1 2^n \leq \underbrace{2^{2n} \leq c_2 2^n}_{2^n}$

$$\underbrace{\frac{2^{\cancel{2n}}}{\cancel{2^n}} \leq \frac{c_2 \cancel{2^n}}{\cancel{2^n}}}$$

$2^n \leq c_2 \Rightarrow$ Impossible since 2^n is bounded by a constant. $\Rightarrow \boxed{\text{False}}$

III. $F(n) \leq c_1 n^2 \Leftrightarrow c_2 n^2 \leq g(n) \leq c_3 n^2$

$c_1 n^2 \leq F(n) * g(n) \leq c_1 n^2 c_3 n^2$

$c_1 n^2 \leq F(n) * g(n) \leq c_1 c_3 n^4$

$n^2 \neq n^4 \Rightarrow \boxed{\text{False}}$

(They must be equal. $\Rightarrow c_1 \underline{h(n)} \leq \dots c_2 \underline{h(n)}$)

5) a) $T(n) = 2T(n/2) + n$, $T(1) = 1$

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

$T(n/2) = 2T(n/4) + n/2 \Rightarrow 2(2T(n/4) + n/2) + n$

$4T(n/4) + n + n$

$T(n/4) = 2T(n/8) + n/4$ $\quad 4T(n/4) + 2n$

$4(2T(n/8) + n/4) + 2n$

$8T(n/8) + n + 2n$

$8T(n/8) + 3n$

$\vdots$

$2^k T(\underbrace{n/2^k}_{1}) + kn$

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

$\underbrace{2^k}_{n} \underbrace{T(1)}_{1} + \log(n).n = n + n.\log(n) \Rightarrow \boxed{O(n.\log(n))}$

b) $T(n) = 2T(n-1) + 1$, $T(0) = 0$

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

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

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

$\vdots$

$= 2^k T(\underbrace{n-k}_{0}) + 2^{k-1} + 2^{k-2} + 2^{k-3} + \dots + 2 + 1$

$n - k = 0 \Rightarrow n = k \Rightarrow \underbrace{2^n \underbrace{T(0)}_{0}}_{0} + \underbrace{2^{k-1} + \dots + 2 + 1}_{2^k - 1 = 2^n - 1 \; (k=n)} = 2^n - 1 = \boxed{O(2^n)}$

6)

Let A.length is n.

Print : Ɵ(1)

If statement : Ɵ(1)

Nested loop : Ɵ(n) * Ɵ(n) = Ɵ(n^2)

Time complexity : Ɵ(n^2)

```java
public static void findPair(int[] A, int sum) {
    for (int i = 0; i < A.length - 1; i++) {
        for (int j = i + 1; j < A.length; j++) {
            if (A[i] + A[j] == sum) {
                System.out.println(A[i] + "," + A[j]);
            }
        }
    }
}
```

```
Size of array is 10
0,5
1,4
2,3
Execution time: 0.0952
*********************************
Size of array is 70
0,5
1,4
2,3
Execution time: 0.1217
*********************************
Size of array is 110
0,5
1,4
2,3
Execution time: 0.1794

----------------------------------
----------------------------------
----------------------------------
RECURSIVE FUNCTİON

Size of array is 10
0,5
1,4
2,3
Execution time: 0.0539
*********************************
Size of array is 70
0,5
1,4
2,3
Execution time: 0.4665
*********************************
Size of array is 110
0,5
1,4
2,3
Execution time: 0.7601
```

Execution time of function increases when array size increases. The bound of for loops is array size that's why it must be increased.

7)

Let A.length is n.

Print : Θ(1)   ;   If statements : Θ(1)   ;   Inside of if : Θ(1)

Best case : Θ(1) -> Base case

Worst case : Θ(n)

Time complexity : O(n)

```
public static void recFindPair(int[] A, int sum, int i, int j) {

    if (i == A.length - 1)
        return;

    else {
        if (A[i] + A[j] == sum)
            System.out.println(A[i] + "," + A[j]);
        if (j == A.length - 1){
            i++;
            j = i;
        }
        recFindPair(A, sum, i, ++j);
    }

}
```