a) Ordenar las siguientes funciones en orden creciente de tiempo de ejecución. Además, para cada una de las funciones $T_1, \ldots, T_5$ determinar su velocidad de crecimiento (expresarla con la notación $O(\cdot)$).

$$\begin{aligned}
T_1 &= n^2 + 2 \cdot 4^n + 5^3 \\
T_2 &= 3 \cdot 2^n + 5n^4 + 2n \\
T_3 &= 2 \log n + \sqrt{n} + 3n^2 + 2n^5 \\
T_4 &= 4^5 + 2.7 \log_4 n + \log_2(5)\, n^{1.5} \\
T_5 &= \log_2 n + 5
\end{aligned}$$

Orden creciente según tiempo de ejecución:

$$T_5 < T_4 < T_3 < T_2 < T_1$$

Complejidades:

$$\begin{aligned}
T_1(n) &\in O(4^n) \\
T_2(n) &\in O(2^n) \\
T_3(n) &\in O(n^5) \\
T_4(n) &\in O(n^{1.5}) \\
T_5(n) &\in O(\log n)
\end{aligned}$$

b) Suponga que cierto procesador tiene una frecuencia de reloj de 3 GHz, y simplifique el análisis suponiendo que por cada tres ciclos de reloj se realiza una operación en particular. Estime de qué tamaño puede ser el problema que se puede resolver en un segundo usando un algoritmo que requiere $T(n)$ operaciones, con los siguientes valores de $T(n)$: $\log(n)$, n, $n\log(n)$, n^2, 2^n y $n!$.

Si el procesador tiene una frecuencia de 3 GHz, y realiza una operación cada tres ciclos de reloj, entonces tiene una capacidad de realizar:

$$\text{Operaciones por segundo} = \frac{3 \times 10^6\,\text{Hz}}{3} = 1 \times 10^6$$

Para calcular el tamaño máximo de la entrada n de un algoritmo que requiere $T(n)$ operaciones en un segundo, planteamos:

$$\begin{aligned}
T(n) &= 1 \times 10^6 \\
n &= T^{-1}(1 \times 10^6)
\end{aligned}$$

Esto, considerando que $T(n)$ tiene inversa. De lo contrario, para calcular el número de operaciones n se utilizará el siguiente algoritmo:

```
def calcular_n(T: callable, operaciones: int) -> int:
    n = 1
    while T(n) < operaciones:
        n += 1
    return n - 1
```

Entonces, para cada algoritmo obtenemos un n_{max}:

Complejidad $T(n)$	$n_{\max}$
$\log n$	$10^{1\,000\,000}$
n	10^6
$n \log n$	189 481
n^2	1 000
2^n	19
$n!$	9

c) Una manera útil de pensar acerca del crecimiento de complejidad computacional es considerar cómo varía el tiempo de cómputo si el tamaño del problema se duplica. Determine el incremento de costo para:

$$T(n) : 1,\ \log(n),\ n,\ n \log(n),\ n^2,\ n^3,\ 2^n.$$

Definimos el incremento I como:

$$T(2n) = I \cdot T(n)$$

Por ende:

$$I(T) = \frac{T(2n)}{T(n)}$$

Aplicando esta definición a cada algoritmo:

$$I(1) = \frac{1}{1} = 1$$

$$I(log(n)) = \frac{log(2n)}{log(n)} = \frac{log(2) + log(n)}{log(n)} = \frac{log(2)}{log(n)} + 1$$

$$I(n) = \frac{2n}{n} = 2$$

$$I(n\,log(n)) = \frac{2n\,log(2n)}{n\,log(n)} = \frac{2\,(log(2) + log(n))}{log(n)} = \frac{2\,log(2)}{log(n)} + 2$$

$$I(n^2) = \frac{(2n)^2}{n^2} = 4$$

$$I(n^3) = \frac{(2n)^3}{n^3} = 9$$

$$I(2^n) = \frac{2^{2n}}{2^n} = \frac{2^2\,2^n}{2^n} = 4$$

d) Considere el siguiente algoritmo:

```
def algoritmo(L, p, x):
    if p == len(L):
        return False
    if L[p] == x:
        return True
    return algoritmo(L, p + 1, x)
```

- Comente qué se implementa.
- Determine el tiempo de ejecución $T(n)$ para:
  - El peor caso $T_{\text{peor}}(n)$
  - El mejor caso $T_{\text{mejor}}(n)$
  - El caso promedio $T_{\text{mean}}(n)$
- Determine el orden de complejidad algorítmica $O(n)$.

El algoritmo anterior implementa la búsqueda de un elemento `x` en una secuencia `L` a partir del índice `p`, En el caso de `x` existir en `L[p:]` entonces se devuelve `True`, caso contrario `False`.

Considerando $n =$ `len(L) - p`. El caso menos favorable se da cuando no existe, en este caso $T_{peor} = n$. El caso menos favorable se da cuando x se encuentra exactamente en `L[p]`, siendo $T_{mejor} = 1$. Finalmente, el caso promedio es $T_{mean} = n/2$.

El orden de complegidad algoritmica es $O(n)$, ya que el tiempo de ejecución aumenta linealmente con el tamaño de la entrada.