Actividad 1

Alumno: **Luis Fernando Izquierdo Berdugo**

Materia: **Análisis de algoritmos y estructuras para datos masivos**

Fecha: **14 de Agosto de 2024**

Introducción

Normalmente, para resolver un problema existen varios algoritmos que podrían ser una solución. Para elegir el mejor camino a tomar, se suele medir la eficiencia de los algoritmos por medio del tiempo de ejecución que estos toman.

> Se puede decir entonces que el algoritmo para un problema requiere un tiempo del orden de $t(n)$ para una función dada t, si existe una constante positiva c y una implementación del algoritmo capaz de resolver todos los casos de tamaño n en un tiempo que no sea superior a $c\,t(n)$ unidades de tiempo. (Bel, 2020, p. 28).

Algunos órdenes se presentan de manera constante en los algoritmos, por lo que ya han sido clasificados en los siguientes:

- De tiempo constante $O(c)$
- De tiempo logarítmico $O(\log n)$
- De tiempo lineal $O(n)$
- De tiempo casi lineal $O(n \log n)$
- De tiempo polinómico $O(n^a)$
- De tiempo exponencial $O(a^n)$
- De tiempo factorial $O(n!)$
- De orden potencial exponencial $O(n^n)$

En esta actividad haremos comparaciones entre estas órdenes de crecimiento para ver cómo se desempeñan unas contra otras.

Instrucciones para la actividad

1. Utilizar el notebook de Jupyter para generar las siguientes comparaciones de ordenes de crecimiento (una figura por comparación, i.e., cinco figuras)

- $O(1)$ vs $O(\log n)$
- $O(n)$ vs $O(n \log n)$
- $O(n^2)$ vs $O(n^3)$
- $O(a^n)$ vs $O(n!)$
- $O(n!)$ vs $O(n^n)$

Escoja los rangos adecuados para cada comparación, ya que como será evidente después, no es práctico fijar los rangos.

1. Haga una tabla donde simule tiempos de ejecución suponiendo que cada operación tiene un costo de 1 micro-segundo:

- Suponga que cada uno de los ordenes de crecimiento anteriores es una expresión que describe el costo de un algoritmo teniendo en cuenta el tamaño de la entrada del algoritmo n.
- Use como los diferentes tamaños de entrada $n=100$; $n=1000$; $n=10000$ y $n=100000$.
- Note que para algunas fórmulas, los números pueden ser muy grandes.

1. Dentro del notebook añada un breve ensayo reflexionando sobre los costos de cómputo necesarios para manipular grandes volumenes de información.

Inciso 1

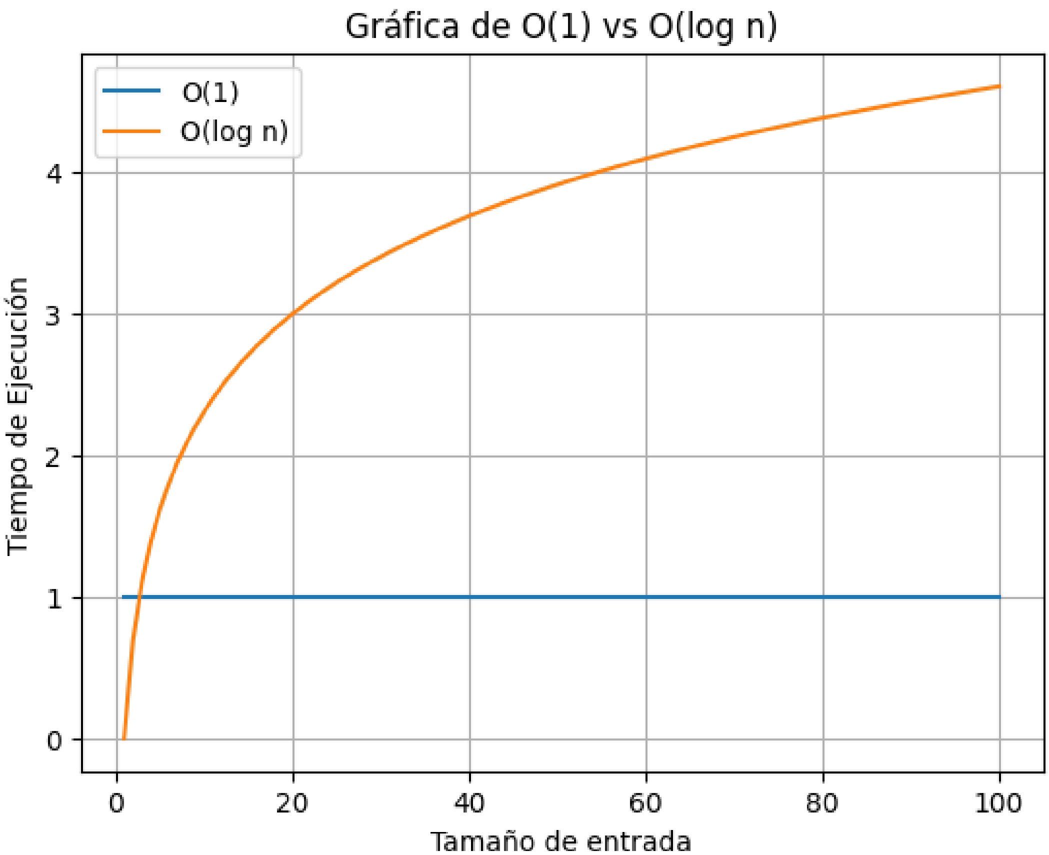
1. **Utilizar el notebook de Jupyter para generar las siguientes comparaciones de ordenes de crecimiento (una figura por comparación, i.e., cinco figuras)**

- $O(1)$ **vs** $O(\log n)$
- $O(n)$ **vs** $O(n \log n)$
- $O(n^2)$ **vs** $O(n^3)$
- $O(a^n)$ **vs** $O(n!)$
- $O(n!)$ **vs** $O(n^n)$

$O(1)$ vs $O(\log n)$

```
import matplotlib.pyplot as plt
import numpy as np

# Definimos el rango de valores para x
x = np.linspace(1, 100, 100)

# Calculamos los valores de y para cada x
y_const = np.ones_like(x)
y_log = np.log(x)

# Creamos la figura y los ejes
plt.figure()
plt.plot(x, y_const, label='O(1)')
plt.plot(x, y_log, label='O(log n)')

# Personalizamos el gráfico (opcional)
plt.title("Gráfica de O(1) vs O(log n)")
plt.xlabel("Tamaño de entrada")
plt.ylabel("Tiempo de Ejecución")
plt.grid(True)
```

```
plt.legend()

# Mostramos el gráfico
plt.show()
```

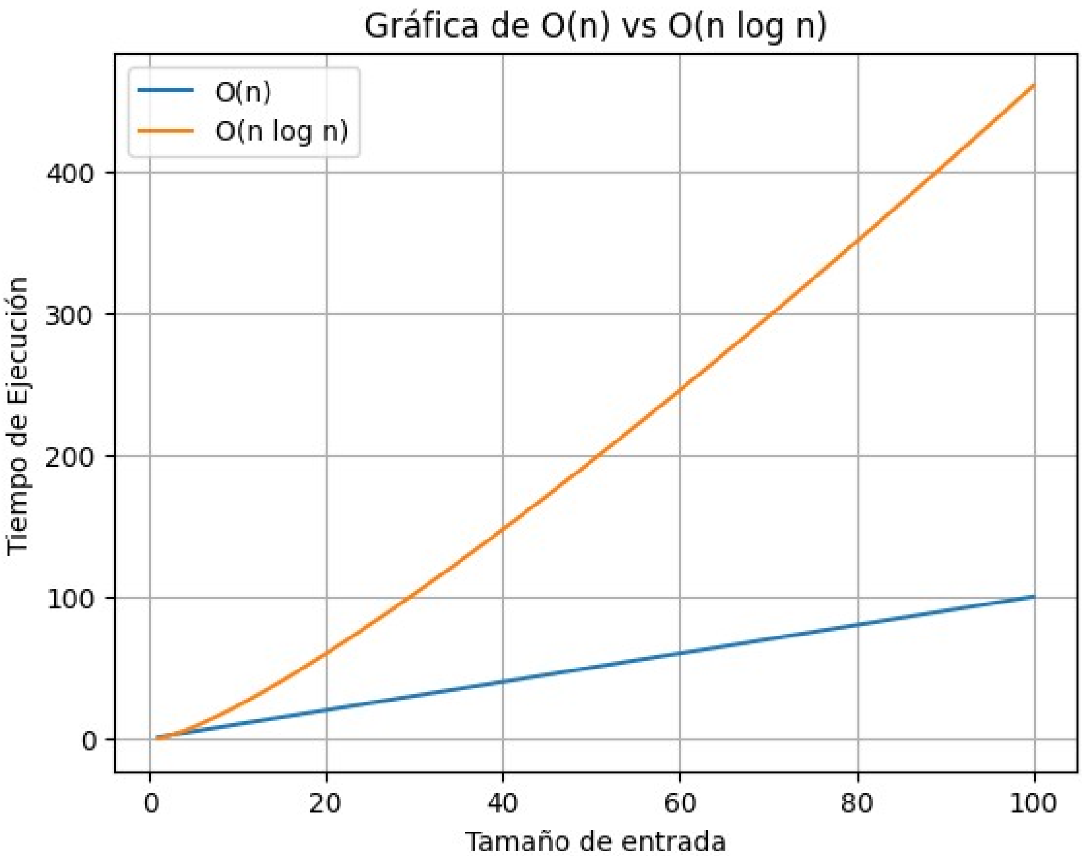


En la figura anterior podemos observar como el tiempo de ejecución para ambos casos difiere mucho. En el caso del tiempo constante vemos como no se afecta por el tamaño de entrada, a diferencia del tiempo logarítmico que aumenta el tiempo de ejecución con el tamaño de entrada

$O(n)$ vs $O(n \log n)$

```
x = np.linspace(1, 100, 100)

y_lineal = x
y_casi = x * np.log(x)

plt.figure()
plt.plot(x, y_lineal, label='O(n)')
plt.plot(x, y_casi, label='O(n log n)')
plt.title("Gráfica de O(n) vs O(n log n)")
plt.xlabel("Tamaño de entrada")
plt.ylabel("Tiempo de Ejecución")
plt.grid(True)
```

```
plt.legend()

plt.show()
```

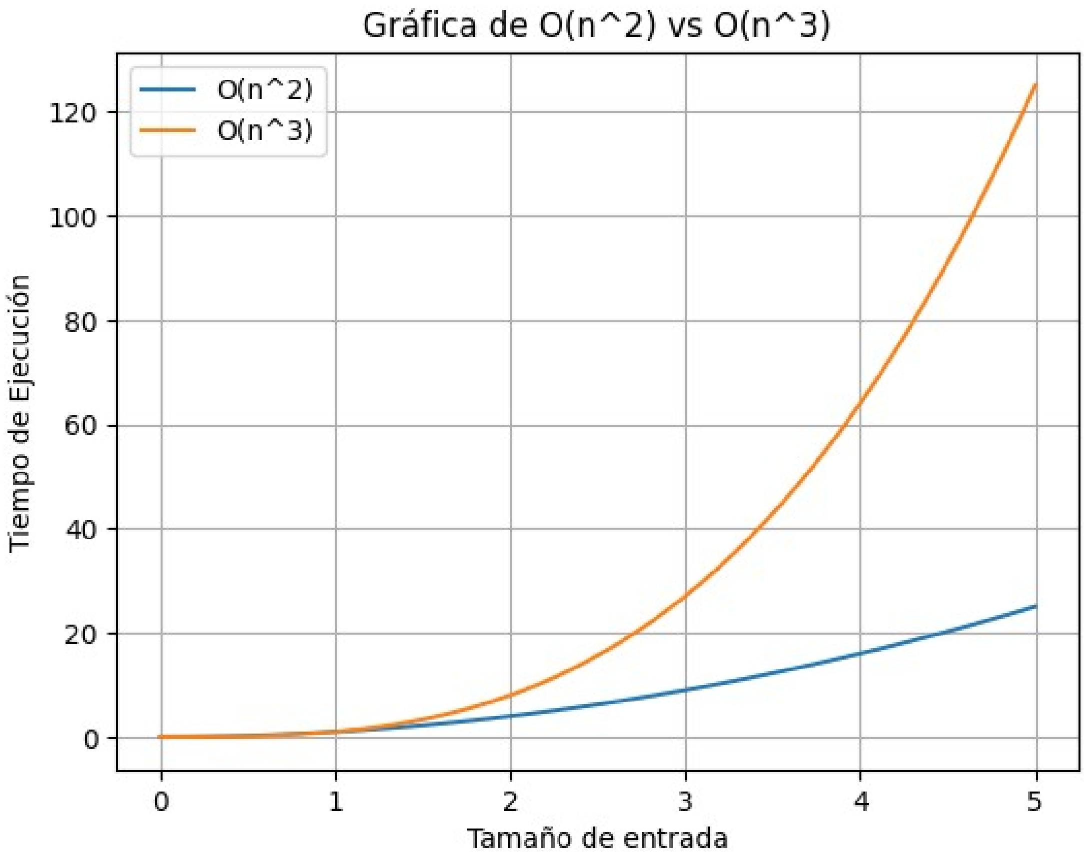


En esta comparación vemos como, a pesar de lucir diferente, si analizamos bien podemos observar que las líneas son muy parecidas, ambos tiempos de ejecución son directamente proporcionales al tamaño de entrada, esto es esperado ya que la pista la tenemos en los nombres: tiempo lineal y tiempo casi lineal.

$O(n^2)$ vs $O(n^3)$

```
x = np.linspace(0, 5, 100)

y_cuad = x**2
y_cub = x**3

plt.figure()
plt.plot(x, y_cuad, label='O(n^2)')
plt.plot(x, y_cub, label='O(n^3)')
plt.title("Gráfica de O(n^2) vs O(n^3)")
plt.xlabel("Tamaño de entrada")
plt.ylabel("Tiempo de Ejecución")
plt.grid(True)
```

```
plt.legend()
plt.show()
```

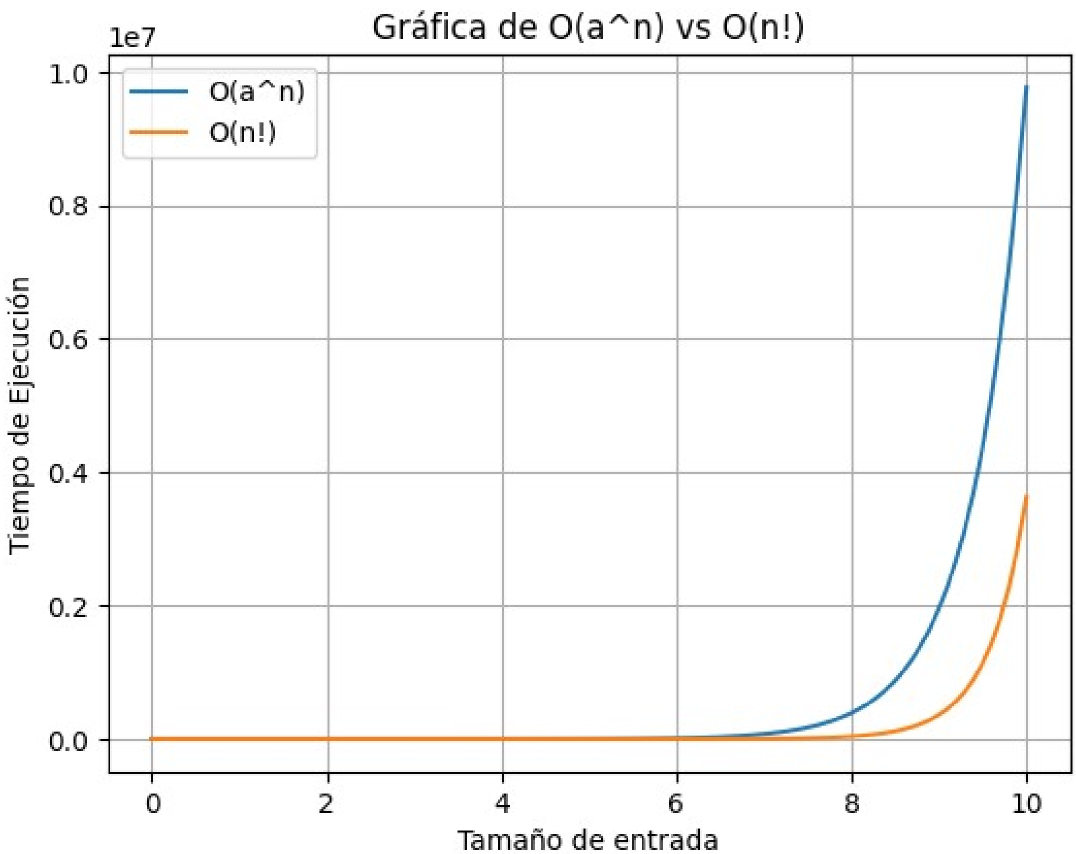


En esta gráfica podemos observar como la operación de tiempo cuadrática parece ser más eficiente que la operación de tiempo cúbica, ya que al aumentar el tamaño de entrada, el tiempo de ejecución no se dispara al mismo nivel que la cúbica, la cual, crece bastante con el mismo tamaño de entrada.

$O(a^n)$ vs $O(n!)$

```
#Importamos la funcion factorial de scipy
from scipy.special import factorial

x = np.linspace(0, 10, 100)

y_exp = 5**x
y_fact = factorial(x)

plt.figure()
plt.plot(x, y_exp, label='O(a^n)')
plt.plot(x, y_fact, label='O(n!)')
plt.title("Gráfica de O(a^n) vs O(n!)")
plt.xlabel("Tamaño de entrada")
```

```
plt.ylabel("Tiempo de Ejecución")
plt.grid(True)
plt.legend()

plt.show()
```

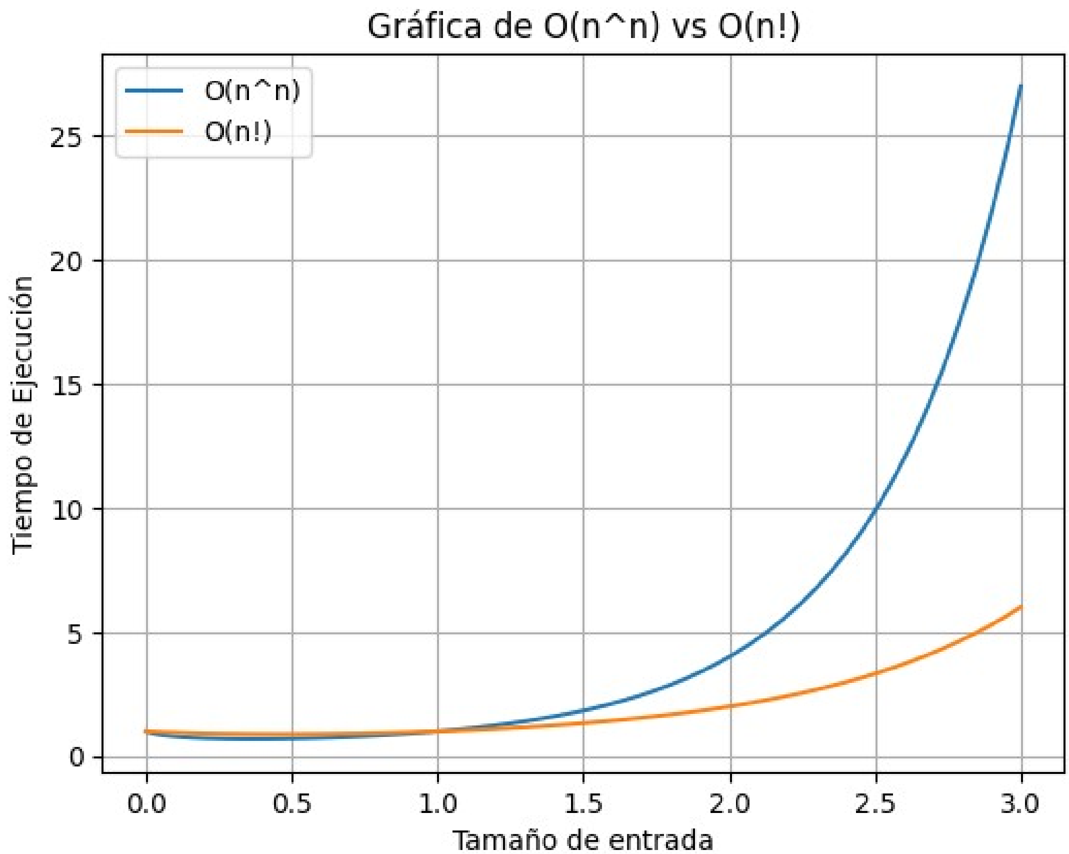


En esta gráfica podemos observar como las funciones parecen ser bastante parecidas, ya que podemos notar como la operación de tiempo factorial comienza a subir de la misma manera que la de tiempo exponencial.

$O(n!)$ vs $O(n^n)$

```
x = np.linspace(0, 3, 100)

y_exp = x**x
y_fact = factorial(x)

plt.figure()
plt.plot(x, y_exp, label='O(n^n)')
plt.plot(x, y_fact, label='O(n!)')
plt.title("Gráfica de O(n^n) vs O(n!)")
plt.xlabel("Tamaño de entrada")
plt.ylabel("Tiempo de Ejecución")
```

```
plt.legend()
plt.grid(True)

plt.show()
```



Gracias a esta gráfica podemos observar que la operación de tiempo factorial, a pesar de ser una de las más tardadas, puede ser muchísimo mejor que aquella de tiempo potencial exponencial, ya que esta última se dispara en tiempo de ejecución con el mismo tamaño de entrada.

Inciso 2

Haga una tabla donde simule tiempos de ejecución suponiendo que cada operación tiene un costo de 1 micro-segundo:

- **Suponga que cada uno de los ordenes de crecimiento anteriores es una expresión que describe el costo de un algoritmo teniendo en cuenta el tamaño de la entrada del algoritmo n.**
- **Use como los diferentes tamaños de entrada** $n=100$; $n=1000$; $n=10000$ **y** $n=100000$**.**
- **Note que para algunas fórmulas, los números pueden ser muy grandes.**

```python
#Importamos pandas para crear la tabla
import pandas as pd

#Creamos las variables de n que serán utilizadas a lo largo del código
n1 = 100
n2 = 1000
n3 = 10000
n4 = 100000

#Creamos el dataframe con las columnas para guardar los datos
df = pd.DataFrame(columns=['Orden','n = 100', 'n = 1000', 'n = 10000',
'n = 100000'], dtype=int)
df
```

```
Empty DataFrame
Columns: [Orden, n = 100, n = 1000, n = 10000, n = 100000]
Index: []
```

Para simular el tiempo de ejecución de las orden de crecimiento, se declarará una función donde se ejecute la acción, para poder ejecutar cada una de las n propuestas. Se guardarán los resultados en un Dataframe de Pandas.

```python
def O_constante(n):
    t = np.ones_like(n)
    return t

const = pd.Series(["O(1)",
                   O_constante(n1),
                   O_constante(n2),
                   O_constante(n3),
                   O_constante(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

```
  Orden n = 100 n = 1000 n = 10000 n = 100000
0  O(1)       1        1         1          1
```

$O(\log n)$

```python
def O_log(n):
    t = np.log(n)
    return t

const = pd.Series(["O(log n)",
                   O_log(n1),
                   O_log(n2),
                   O_log(n3),
                   O_log(n4)
                   ], index=df.columns)
```

```python
df.loc[len(df)] = const
df
```

	Orden	n = 100	n = 1000	n = 10000	n = 100000
0	O(1)	1	1	1	1
1	O(log n)	4.60517	6.907755	9.21034	11.512925

$O(n)$

```python
def O_lineal(n):
    t = n
    return t

const = pd.Series(["O(n)",
                   O_lineal(n1),
                   O_lineal(n2),
                   O_lineal(n3),
                   O_lineal(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

	Orden	n = 100	n = 1000	n = 10000	n = 100000
0	O(1)	1	1	1	1
1	O(log n)	4.60517	6.907755	9.21034	11.512925
2	O(n)	100	1000	10000	100000

$O(n \log n)$

```python
def O_clineal(n):
    t = n * np.log(n)
    return t

const = pd.Series(["O(nlogn)",
                   O_clineal(n1),
                   O_clineal(n2),
                   O_clineal(n3),
                   O_clineal(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

	Orden	n = 100	n = 1000	n = 10000	n = 100000
0	O(1)	1	1	1	1
1	O(log n)	4.60517	6.907755	9.21034	11.512925
2	O(n)	100	1000	10000	100000
3	O(nlogn)	460.517019	6907.755279	92103.40372	1151292.546497

$O\left(n^2\right)$

```
def O_cuad(n):
    t = n ** 2
    return t

const = pd.Series(["O(n^2)",
                   O_cuad(n1),
                   O_cuad(n2),
                   O_cuad(n3),
                   O_cuad(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

	Orden	n = 100	n = 1000	n = 10000	n = 100000
0	O(1)	1	1	1	1
1	O(log n)	4.60517	6.907755	9.21034	11.512925
2	O(n)	100	1000	10000	100000
3	O(nlogn)	460.517019	6907.755279	92103.40372	1151292.546497
4	O(n^2)	10000	1000000	100000000	10000000000

$O\left(n^3\right)$

```
def O_cub(n):
    t = n ** 3
    return t

const = pd.Series(["O(n^3)",
                   O_cub(n1),
                   O_cub(n2),
                   O_cub(n3),
                   O_cub(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

	Orden	n = 100	n = 1000	n = 10000	n = 100000
0	O(1)	1	1	1	1
1	O(log n)	4.60517	6.907755	9.21034	11.512925
2	O(n)	100	1000	10000	100000
3	O(nlogn)	460.517019	6907.755279	92103.40372	1151292.546497
4	O(n^2)	10000	1000000	100000000	10000000000
5	O(n^3)	1000000	1000000000	1000000000000	1000000000000000

$O\left(a^n\right)$

En esta función se utiliza el módulo "decimal" que implementa precisión hasta con 28 dígitos.

```
import decimal

def O_exp(n):
    t = decimal.Decimal(2) ** n
    return t

const = pd.Series(["O(2^n)",
                   O_exp(n1),
                   O_exp(n2),
                   O_exp(n3),
                   O_exp(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

```
      Orden                             n = 100  \
0      O(1)                                   1
1  O(log n)                             4.60517
2      O(n)                                 100
3  O(nlogn)                          460.517019
4    O(n^2)                               10000
5    O(n^3)                             1000000
6    O(2^n)  1.267650600228229401496703205E+30

                             n = 1000                            n =
10000  \
0                                   1
1
1                            6.907755
9.21034
2                                1000
10000
3                         6907.755279
92103.40372
4                             1000000
100000000
5                          1000000000
1000000000000
6  1.071508607186267320948425049E+301
1.995063116880758384883742163E+3010

                              n = 100000
0                                      1
1                              11.512925
2                                 100000
3                         1151292.546497
4                            10000000000
5                       1000000000000000
6  9.990020930143845079440327643E+30102
```

$O(n!)$

En esta función se usa el módulo "gmpy2" que resuelve operaciones aritméticas con alta precisión, de no usar este módulo, Python marcaría los resultados de $1000!$, $10000!$ y $100000!$ como infinitos, lo cual no es un resultado real.

```
import gmpy2

def O_fact(n):
    t = gmpy2.factorial(n)
    print(t)
    return t

const = pd.Series(["O(n!)",
                   O_fact(n1),
                   O_fact(n2),
                   O_fact(n3),
                   O_fact(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

```
9.3326215443944151e+157
4.0238726007709379e+2567
2.8462596809170545e+35659
2.8242294079603476e+456573
```

```
      Orden                            n = 100  \
0      O(1)                                  1
1  O(log n)                            4.60517
2      O(n)                                100
3  O(nlogn)                         460.517019
4    O(n^2)                              10000
5    O(n^3)                            1000000
6    O(2^n)  1.267650600228229401496703205E+30
7     O(n!)            9.3326215443944151e+157
```

```
                             n = 1000                            n =
10000  \
0                                   1
1
1                            6.907755
9.21034
2                                1000
10000
3                         6907.755279
92103.40372
4                             1000000
100000000
5                          1000000000
```

```
1000000000000
6  1.071508607186267320948425049E+301
1.995063116880758384883742163E+3010
7            4.0238726007709379e+2567
2.8462596809170545e+35659

                             n = 100000
0                                     1
1                             11.512925
2                                100000
3                        1151292.546497
4                           10000000000
5                      1000000000000000
6  9.990020930143845079440327643E+30102
7             2.8242294079603476e+456573
```

$O\left(n^n\right)$

```
def O_potexp(n):
    n = decimal.Decimal(n)
    t = n**n
    print(t)
    return t

const = pd.Series(["O(n^n)",
                   O_potexp(n1),
                   O_potexp(n2),
                   O_potexp(n3),
                   O_potexp(n4)
                   ], index=df.columns)
df.loc[len(df)] = const
df
```

```
1.000000000000000000000000000E+200
1.000000000000000000000000000E+3000
1.000000000000000000000000000E+40000
1.000000000000000000000000000E+500000

      Orden                              n = 100  \
0      O(1)                                    1
1  O(log n)                              4.60517
2      O(n)                                  100
3  O(nlogn)                           460.517019
4    O(n^2)                                10000
5    O(n^3)                              1000000
6    O(2^n)   1.267650600228229401496703205E+30
7     O(n!)              9.332621544394415e+157
8    O(n^n)  1.000000000000000000000000000E+200

                              n = 1000                                 n =
```

```
10000  \
0                                1
1
1                         6.907755
9.21034
2                             1000
10000
3                      6907.755279
92103.40372
4                          1000000
100000000
5                       1000000000
1000000000000
6   1.071508607186267320948425049E+301
1.995063116880758384883742163E+3010
7            4.0238726007709379e+2567
2.8462596809170545e+35659
8  1.000000000000000000000000000E+3000
1.000000000000000000000000000E+40000

                              n = 100000
0                                    1
1                            11.512925
2                               100000
3                       1151292.546497
4                          10000000000
5                     1000000000000000
6   9.990020930143845079440327643E+30102
7            2.8242294079603476e+456573
8  1.000000000000000000000000000E+500000
```

Orden	n = 100	n = 1000	n = 10000	n = 100000
O(1)	1	1	1	1
O(log n)	4.60517	6.907755	9.21034	11.512925
O(n)	100	1000	10000	100000
O(nlogn)	460.517019	6907.755279	92103.40372	1151292.546497
O(n^2)	10000	1000000	100000000	10000000000
O(n^3)	1000000	1000000000	1000000000000	1000000000000000
O(2^n)	1.267650600228229401496703205E+30	1.071508607186267320948425049E+301	1.995063116880758384883742163E+3010	9.990020930143845079440327643E+30102
O(n!)	9.3326215443944151e+157	4.0238726007709379e+2567	2.8462596809170545e+35659	2.8242294079603476e+456573
O(n^n)	1.000000000000000000000000000E+200	1.000000000000000000000000000E+3000	1.000000000000000000000000000E+40000	1.000000000000000000000000000E+500000

En la tabla anterior podemos observar los resultados de procesamiento de las órdenes de crecimiento propuestas con los distintos valores de n.

Entre los resultados podemos destacar el simil entre la operación logarítmica $O(log\, n)$ y la casi lineal $O(n\, log\, n)$, que pareciera ser el mismo resultado multiplicado. De igual manera, vemos el crecimiento en aquellas que tienen como base de una potencia al valor de entrada n, siendo estas la lineal $O(n)$, la cuadrática $O(n^2)$, la cúbica $O(n^3)$ y la potencial exponencial $O(n^n)$.

Ciertamente destacan los valores de las órdenes exponencial $O(C^n)$, factorial $O(n!)$ y potencial exponencial $O(n^n)$, ya que, se llegan a obtener tiempos de ejecución muy altos incluso con valores de entrada n no tan elevados como podría ser $n=100$.

Inciso 3

Dentro del notebook añada un breve ensayo reflexionando sobre los costos de cómputo necesarios para manipular grandes volumenes de información.

Gracias al trabajo realizado en esta actividad, se pudo obtener una idea específica del costo de procesamiento de operaciones sencillas que se pueden presentar en diferentes algoritmos de procesamiento de información. Definitivamente se demostró que las operaciones de tiempo constante son las menos costosas, mientras que aquellas de tiempo factorial y potencial exponencial son las que más consumo tendrían, haciéndolas las menos eficientes para trabajar.

En las gráficas de comparación se pudo observar como algunas no se parecían en nada, como la de tiempo constante $O(1)$ y tiempo logarítmico $O(log n)$, en cambio otras eran bastante parecidas como la de tiempo lineal $O(n)$ y tiempo casi lineal $O(n log n)$.

Es interesante analizar los resultados de la tabla generada con los valores de $n=100, 1000, 10000$ y 100000 ya que nos presentan la gran diferencia que puede existir entre operaciones que son aparentemente parecidas en procesamiento, sin embargo las funciones de tiempo exponencial, factorial y potencial exponencial siguen siendo demasiado grandes para las unidades de tiempo conocidas, llegando a ser el exponencial más de **9 billones de milenios** en el valor de $n=100$, lo cual tardaría más que la existencia de la humanidad.

Si se comparan los resultados de lo que se observó en las gráficas y los resultados de la tabla, se pueden encontrar discrepancias, por ejemplo con la función lineal $O(n)$ y la casi lineal $O(n log n)$, que se esperaba tuvieran resultados similares o mínimo se pueda encontrar una relación evidente entre ambos. También se puede observar que todos aquellos que implicaban una potencia con base n (cuadrática, cúbica, potencial exponencial) tienen una forma de curva bastante parecida y, al analizar los datos de la tabla, se puede encontrar facilmente una relación entre ellas, lo cual se esperaba debido a que todas terminan siendo múltiplos de la misma bas n.

Entre las problemáticas encontradas durante el desarrollo de esta actividad se encontró:

- Hallar el tamaño de entrada correcto para la graficación de las distintas comparaciones, ya que en algunos casos la visualización no era la correcta debido a este factor.
- Manejo de caracteres mayores a los que Python tiene limitados, por lo que se tuvo que buscar una alternativa con el módulo “decimal”, con el cual se puede manejar hasta 28 dígitos.
- Incorrecto cálculo en algunas operaciones, como la operación factorial que para valores de n mayores a 1000 mostraba el resultado como infinito. Esto se pudo corregir gracias al módulo “gmpy2” que permite trabajar con enteros y números racionales de cualquier tamaño, usando su función factorial se pudo obtener un resultado preciso.

A grandes rasgos, esta actividad sirvió para poder tener conciencia del tiempo de ejecución, el cual forma parte importante para el cálculo de eficiencia de un algoritmo y podría ser de utilidad al analizar el costo de almacenamiento, ya que, a veces podría ser mejor tomar un algoritmo cuyo costo de almacenamiento sea menor, pero su tiempo de ejecución sea menor.

Bibliografía

Bel, W. (2020). *Algoritmos y estructuras de datos en Python: un enfoque ágil y estructurado* (1st ed.) [Libro digital, PDF]. Editorial Uader.