



ANALISIS DE ALGORITMOS DE ORDENAMIENTO

Programacion I

Alumnos: Pérez Lucio Gabriel – perezluciogabriel@gmail.com

Profesor/a: Nicolas Quiroz

Fecha de Entrega: 05/06/2025

Lucio Gabriel P.
perezluciogabriel@gmail.com

1. Introducción:

En el campo de la Programación y el desarrollo de Software, la eficiencia con la que se manipulan y se organizan los datos es un factor crucial que impacta directamente en el rendimiento de las aplicaciones, es por eso que, dentro de este contexto, decidimos investigar sobre la eficiencia de los algoritmos de ordenamiento ya que ocupan un lugar central para reorganizar conjuntos de datos, facilitando búsquedas, análisis y operaciones posteriores.

El presente trabajo, se enfoca en la investigación aplicada de distintos algoritmos de ordenamiento, analizando sus principios teóricos, su eficiencia computacional y su comportamiento en distintos escenarios. Además se incluirá un análisis comparativo en notación Big-O, evaluando ventajas, desventajas y casos de uso recomendados. De esta manera, se integran conceptos fundamentales de programación con herramientas de análisis de algoritmos, promoviendo una visión crítica y aplicada sobre las decisiones que los programadores deben tomar al elegir un método de ordenamiento.

2. Marco Teórico

¿Qué es “ordenar”? ¿Para qué sirve en informática?

Según el Diccionario de la Lengua Española, “ordenar” significa:

“tr. Colocar algo o a alguien de acuerdo con un plan o de modo conveniente. *Ordena los recibos por fecha.*”

En pocas palabras es colocar algo bajo un determinado orden ya sea por un plan o de manera conveniente. En Informática, esto es especialmente útil, ya que estamos todo el tiempo trabajando con datos y, a medida que estos van incrementando en cantidad, se hace cada vez más difícil mantener el orden. Por lo que ordenar en el ámbito de la informática refiere a reorganizar los elementos de una estructura de datos (como Listas o Arrays) para que sigan un orden lógico, como por ejemplo:

De menor a mayor:

- [3, 1, 4, 2, 0] → [0, 1, 2, 3, 4]

De mayor a menor:

- [1, 3, 2, 0, 4] → [4, 3, 2, 1, 0]

Alfabéticamente:

- [“Lucio”, “Fernando”, “Rodrigo”, “Marcos”] → [“Fernando”, “Lucio”, “Marcos”, “Rodrigo”]

Ordenamiento por clave:

- Ejemplo: Ordenar por precio en una lista de productos, o por apellido en una lista de alumnos.

Ordenar en informática tiene una importancia crucial en la mayoría de los sistemas y es útil por múltiples razones. Para empezar, **facilita búsquedas**, muchas de las técnicas de búsqueda más eficientes necesitan de datos ordenados ya que buscar un elemento en una lista correctamente ordenada es mucho más rápido que buscar ese mismo elemento pero en una lista que está completamente desordenada o en una disposición no muy conveniente. Además también optimiza el procesamiento de los datos, algunos algoritmos necesitan trabajar con datos en cierto orden para ser eficientes, ejemplo: IA, el ordenamiento suele ser paso previo en muchos procesos de machine learning. Otra razón es la mejora de la presentación de la información para el usuario mismo, es más fácil leer listas ordenadas de productos por ejemplo. También permite comparaciones más claras al graficar o analizar tendencias, como por ejemplo en un software de información en una organización analizando las tendencias del mercado o de sus propias ventas. Ordenar no es sólo una cuestión estética, es una operación fundamental que permite mejorar el rendimiento, la organización y la utilidad de los datos.

Sin algoritmos de ordenamiento, la informática moderna perdería una de sus herramientas más fundamentales. Las bases de datos no podrían listar resultados por fecha o por nombre, un e-commerce (sitio web de comercio electrónico) no podría ordenar productos por precio o popularidad. Por esta razón el estudio de los algoritmos de ordenamiento y su eficiencia no sólo es relevante desde el punto de vista académico, sino que también esencial en el diseño de software robusto, escalable y eficiente.

Existen distintas formas de ordenar una colección de datos, dependiendo del criterio que se quiera aplicar:

- Ordenamiento ascendente: Los elementos se disponen del menor al mayor. Ejemplo: [1, 2, 3, 4]
- Ordenamiento descendente: Los elementos se disponen del mayor al menor. Ejemplo: [4, 3, 2, 1]
- Ordenamiento por clave: En estructuras más complejas se utiliza un atributo específico para ordenar, como por ejemplo un precio o un apellido, tal como habíamos mencionado antes.

Antes de comenzar con los tipos de algoritmos de ordenamiento, es pertinente empezar explicando **qué es el análisis de algoritmos**, ya que estaremos utilizando conceptos del mismo.

El análisis de algoritmos es aquella rama de la informática que se encarga de estudiar la eficiencia de los algoritmos, principalmente en términos de tiempos de ejecución y uso de memoria.

Cuando analizamos un algoritmo, muchas veces queremos saber cuánto tiempo tardará en ejecutarse o cuánta memoria va a necesitar según el tamaño de la entrada. Para eso, se usan tres formas principales de análisis: el mejor caso, el peor caso y el caso promedio.

El **peor caso** representa la situación en la que el algoritmo tarda más en completarse. Es decir, es la ejecución más lenta posible que puede ocurrir con una entrada de cierto tamaño, garantizando que el algoritmo nunca será más lento que eso. Por ejemplo: puede ser que la lista esté en orden inverso.

El **mejor caso** describe la situación más favorable para el algoritmo: cuando se ejecuta en el menor tiempo posible. Ejemplo: puede ser que la lista ya esté ordenada.

El **caso promedio** busca representar lo que sucede normalmente, teniendo en cuenta todas las posibles entradas y su probabilidad. Es el tiempo de ejecución esperado si ejecutáramos el algoritmo muchas veces con distintos datos.

Tipos de eficiencia:

- **Complejidad temporal o Time Complexity:** Evalúa cuánto tiempo toma la ejecución de un algoritmo en función del tamaño de entrada (n). Se expresa con notación Big O:

Ej: $O(n)$, $O(n^2)$, $O(\log n)$, etc.

La notación Big-O se utiliza para describir el comportamiento asintótico de un algoritmo, es decir, cómo crece el tiempo de ejecución (o el uso de memoria) a medida que aumenta el tamaño de la entrada n .

- **Complejidad Espacial:** Mide cuánta memoria adicional necesita el algoritmo, además del espacio que ocupa la entrada.

Existen dos tipos de análisis de algoritmos:

- **Análisis teórico:** Estudia el comportamiento matemático del algoritmo, sin ejecutarlo
- **Análisis empírico:** Mide el tiempo de ejecución real, por ejemplo, utilizando herramientas como `timeit` en el caso de Python. Es el tipo de análisis al que nos centraremos en este proyecto.

El propósito del análisis de algoritmos no es sólo saber si funcionan, sino saber qué tan bien lo hacen, especialmente cuando los datos aumentan en cantidad. Es esencial para el diseño de software porque permite comparar algoritmos que resuelven el mismo problema y ayuda a elegir la solución más adecuada para distintos contextos y situaciones.

Principales algoritmos de ordenamiento:

Un *algoritmo de ordenamiento* es un procedimiento o conjunto de instrucciones que se utilizan para organizar una colección de elementos en un orden específico conveniente. Existen diversos tipos de algoritmos para ordenar datos, cada uno con sus ventajas y desventajas según el tipo de dato, la cantidad de elementos y los recursos disponibles (memoria, capacidad de procesamiento, etc). A continuación, se presentan los algoritmos más representativos, con su funcionamiento básico, análisis de eficiencia y casos de uso más comunes:

1. Ordenamiento de Burbuja (Bubble Sort):

Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Repite este proceso hasta que la lista esté ordenada. Su uso principalmente radica en la simpleza

y su utilización en la educación, ya que es una muestra muy sencilla de como funciona un algoritmo de ordenamiento, pero su eficiencia es muy baja y no suele ser muy utilizado en la industria del software. A medida que la cantidad de datos crece, su eficiencia es cada vez menor y menor. Se utiliza para trabajar con listas pequeñas.

Código:

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j in range(0, n - i - 1):  
            if arr[j] > arr[j + 1]:  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Complejidad:

- Mejor caso: $O(n)$ (lista ya ordenada)
- Peor caso: $O(n^2)$

El mejor caso es cuando la lista ya está ordenada y el peor caso es cuando la lista se encuentra en el orden inverso al criterio utilizado.

2. Ordenamiento de inserción (*insertion sort*):

Lo que hace este algoritmo es tomar cada elemento e insertarlo en la posición correcta respecto a los elementos anteriores. Supongamos que queremos ordenar de menor a mayor, este algoritmo iría número a número, comparando hacia atrás hasta encontrar el primer número menor que el que está intentando insertar, y lo colocaría justo después.

Es un algoritmo muy intuitivo y sencillo, tanto que, por ejemplo, es el que utilizamos para ordenar las cartas mientras jugamos o el que utilizaríamos para ordenar una pila de libros por su año de lanzamiento. Iríamos uno por uno, verificaríamos su año de lanzamiento y lo colocaríamos en su lugar correspondiente. Para listas pequeñas o casi ordenadas funciona muy bien, pero a medida que vamos subiendo la cantidad de datos a ordenar, su eficiencia comienza a caer.

Código:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

Complejidad:

- Mejor caso: $O(n)$
- Peor caso: $O(n^2)$

El mejor caso es cuando la lista ya está ordenada y el peor, es cuando la lista se encuentra en el orden inverso.

3. Ordenamiento de Selección (Selection Sort):

Lo que este algoritmo hace es recorrer la totalidad de la lista a ordenar, buscar el elemento más pequeño y colocarlo en la primer posición. Luego, repite este proceso con el segundo elemento, colocándolo en la segunda posición y así de esta misma forma es como logra ordenar toda la lista.

Código:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        # Encontrar el índice del elemento mínimo
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Intercambiar el elemento mínimo con el elemento actual
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

Al igual que los algoritmos anteriormente mencionados, Selection Sort también es fácil de entender y eficiente a pequeñas escalas o cuando la lista está casi ordenada, sin embargo es lento para listas más grandes.

Complejidad:

- Peor caso: $O(n^2)$, ya que siempre realiza comparaciones para encontrar el mínimo.
- Mejor caso: $O(n^2)$, incluso si la lista está ordenada.

4. Ordenamiento rápido (QuickSort):

Quick sort es un algoritmo recursivo cuyo modus operandi es seleccionar un pivote, dividir la lista en dos, colocar por un lado aquellos elementos mayores que el pivote y por otro lado, aquellos que sean menores que el pivote, luego se aplica recursivamente a cada sublista. En cuanto a eficiencia, Quicksort suele ser más rápido para listas grandes que los algoritmos que hemos visto hasta ahora (selection, insertion y bubble sort), dependiendo de la forma en la que se aplica.

Código:

```
def quicksort(arr):  
    if len(arr) <= 1:  
        return arr  
    else:  
        pivot = arr[0]  
        less = [x for x in arr[1:] if x <= pivot]  
        greater = [x for x in arr[1:] if x > pivot]  
        return quicksort(less) + [pivot] + quicksort(greater)
```

Complejidad:

- Peor caso: $O(n^2)$, cuando el pivote seleccionado es el menor o mayor elemento en cada partición (por ejemplo, si la lista ya está ordenada).
- Mejor caso: $O(n \log n)$, cuando el pivote divide la lista en dos partes aproximadamente iguales.
- Caso promedio: $O(n \log n)$.

5. Sorted() - Función de Python

Adicionalmente, en el presente trabajo también comprobaremos y compararemos con el resto de algoritmos la función Sorted() de Python.

Python implementa su función sorted() utilizando el algoritmo Timsort, escrito en C dentro del núcleo del intérprete. Esta elección se debe a su eficiencia en casos reales, su estabilidad y su buen rendimiento en listas parcialmente ordenadas.

Complejidad:

- Mejor caso: $O(n)$
- Peor caso: $O(n \log n)$
- Promedio $O(n \log n)$

Comparativa de Complejidades:

Algoritmo	Mejor caso	Peor caso	Promedio
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
sorted()	$O(n)$	$O(n \log n)$	$O(n \log n)$

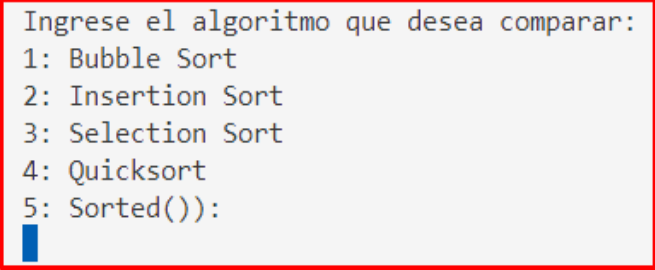
3. Caso Práctico

Para el caso práctico del presente trabajo decidimos hacer un programa en python en el que el usuario sea capaz de poder elegir 2 algoritmos presentes en este trabajo y que desee comparar y que devuelva un gráfico de líneas simple con la comparación de ambos.

Para empezar, pasaremos a explicar como funciona el programa ya terminado y luego iremos bloque a bloque explicando qué es lo que hace cada uno.

Lo primero que hace el programa cuando se ejecuta es preguntarle al usuario cuál es el primer algoritmo que desea comparar, luego se le pregunta por el segundo algoritmo con una interfaz similar:

```
22
23 def selection_sort(arr):
24     n = len(arr)
25     for i in range(n):
```



PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS GITLENS

PS C:\Users\lgp_2\Desktop\UTN programacion\CARRERA\Programacion_1\integra

Ingrese el algoritmo que desea comparar:

- 1: Bubble Sort
- 2: Insertion Sort
- 3: Selection Sort
- 4: Quicksort
- 5: Sorted():

En este menú el usuario puede elegir, por ejemplo, el algoritmo de ordenamiento Bubble Sort, colocando el número 1 por consola, así también el algoritmo Quicksort con el número 4.

Una vez elegidos ambos algoritmos, el programa nos pregunta en qué contexto queremos probar dichos algoritmos, si elegimos la opción 1 los compararemos en un contexto de listas pequeñas, de 100 a 1000 elementos. Los algoritmos serán llamados a ordenar listas de 100, 500 y 1000 elementos, y se medirá el tiempo en el que completen la tarea. De la misma forma si elegimos la opción 2, con la diferencia de que serán probados en un contexto de listas grandes, donde tendran que ordenar listas de 1.000, 5.000 y 10.000 elementos.

Por motivos de prueba, voy a elegir comparar Bubble Sort e Insertion Sort, la opción 1 y 2 respectivamente. Además elegiré probar los algoritmos en un contexto de **listas grandes y pequeñas**:

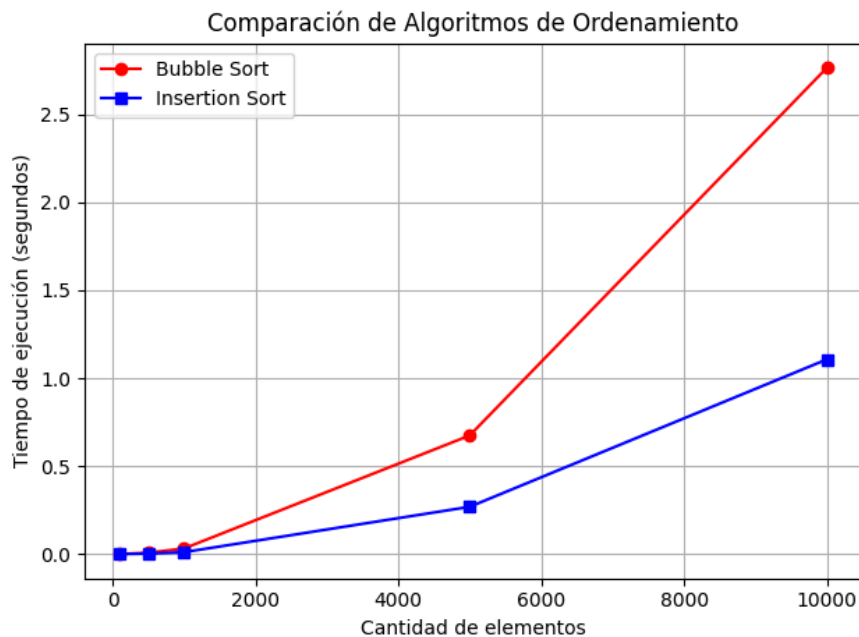
```

Ingrese el algoritmo que desea comparar:
1: Bubble Sort
2: Insertion Sort
3: Selection Sort
4: Quicksort
5: Sorted():
1
Ingrese el segundo algoritmo que desea comparar:
1: Bubble Sort
2: Insertion Sort
3: Selection Sort
4: Quicksort
5: Sorted():
2
¿En qué listas quiere probar los algoritmos?
1: Listas pequeñas (de 100 a 1000 elementos)
2: Listas grandes y pequeñas (de 100 a 10.000 elementos)
2

```

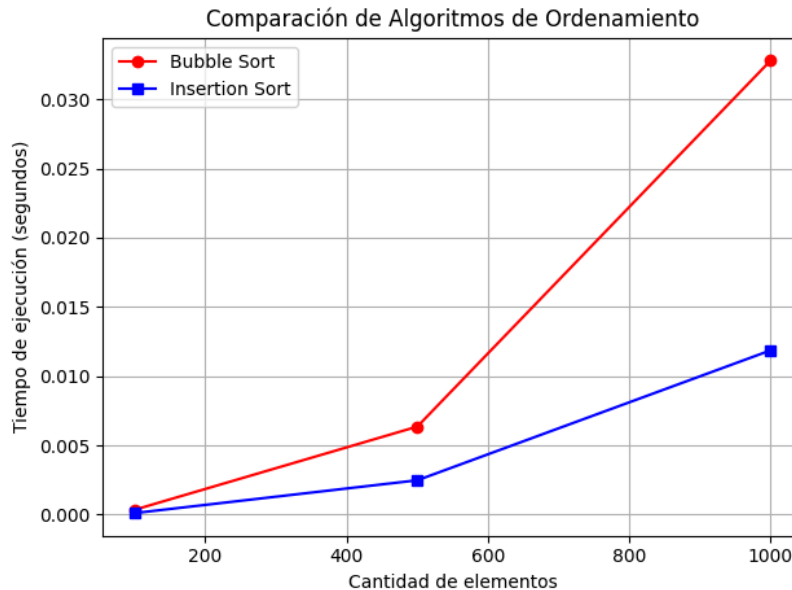
Una vez ingresadas las opciones, el programa procederá a generar un gráfico simple de comparación de ambos algoritmos, cuyo eje X representa la cantidad de elementos de las listas con las que se testearon los algoritmos y el eje Y la cantidad de segundos que tardó el algoritmo en ordenar dichas listas. La línea roja es el primer algoritmo que el usuario eligió y la línea azul es el segundo.

Aquí dejamos un ejemplo del gráfico generado en esta prueba:



En el mismo, podemos observar la diferencia que hay entre el rendimiento de Bubble Sort y Insertion Sort. Esto es, probablemente, porque elegimos que la prueba sea en un contexto de listas grandes de hasta 10.000 elementos, como se puede ver en el eje X.

Si repetimos el proceso, pero esta vez, en el contexto, elegimos la opción de listas pequeñas, obtendremos una versión del gráfico más a detalle en la zona de 100 y 1000 elementos:



Entonces podemos notar que a medida que vamos subiendo de cantidad de elementos, cada vez se hace más difícil, especialmente para Bubble Sort, ordenar los elementos.

Una vez cerrado el gráfico, el programa imprime por pantalla los resultados en segundos, de manera escrita. Si elegimos mostrar los resultados de listas pequeñas, imprimirá 3 resultados, correspondientes a listas de 100, 500, y 1000 elementos.

```
¿En qué listas quiere probar los algoritmos?  
1: Listas pequeñas (de 100 a 1000 elementos)  
2: Listas grandes y pequeñas (de 100 a 10.000 elementos)  
1  
Resultados de Bubble Sort: (0.00035, 0.00802, 0.02708)  
Resultados de Insertion Sort: (0.00012, 0.00246, 0.01059)
```

Explicación del programa paso a paso:

En las primeras líneas de código comenzamos importando las librerías necesarias para el proyecto, que son matplotlib.pyplot, importada como "plt". Esta librería es la que utilizamos para graficar las comparaciones entre los algoritmos, es la encargada de hacer el gráfico de líneas en este caso. Realmente es muy sencilla de utilizar y fue muy útil en la realización del presente proyecto. Luego, se importaron dos librerías nativas de Python, random, que se utiliza para generar números

aleatorios y `timeit` que es utilizada en análisis de algoritmos para poder medir el tiempo de ejecución de los algoritmos.

```
import matplotlib.pyplot as plt
import random
import timeit
```

Una vez importadas las librerías, el código sigue con las definiciones de los algoritmos de ordenamientos y algunas funciones:

```
#algoritmos de ordenamiento ↓
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

def insertion_sort(lista):
    for i in range(1, len(lista)):
        actual = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > actual:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        # Encontrar el índice del elemento mínimo
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Intercambiar el elemento mínimo con el elemento actual
        arr[i], arr[min_index] = arr[min_index], arr[i]

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

Una de las funciones que se declaran es la de “medidor_de_tiempos(funcion, lista)” y básicamente lo que hace es crear una copia de la lista (ya que daba error en quicksort por la recursión, ya que quicksort no es in situ), luego comienza la medición del tiempo, almacena en “resultado” la ejecución de la función pasada por parámetro (alguna de las anteriormente definidas, elegida luego por el usuario), luego finaliza el contador de tiempo y retorna el tiempo inicial menos el tiempo final para calcular el tiempo que le tomó al algoritmo elegido, ordenar la lista en cuestión.

```
#medicion de tiempos
def medidor_de_tiempos(funcion, lista):
    lista_copia = lista.copy()
    tiempo_inicial = timeit.default_timer()
    resultado = funcion(lista_copia)
    tiempo_final = timeit.default_timer()
    return tiempo_final - tiempo_inicial
```

Luego definimos la función “graficar()”, la cual utiliza la librería matplotlib para graficar en pantalla el resultado de la comparación. Básicamente lo que hace es primero graficar dos líneas con la función “plot()” a la cual se le pasa como parámetro el “tamaño” que es básicamente el eje x, contiene por ejemplo: “[100, 500, 1000]”. Luego se le pasa “tiempos_eleccion1” que básicamente son los tiempos que le tomó al primer algoritmo elegido por el usuario, ordenar las diferentes listas, así también “tiempos_eleccion2” son los tiempos que le tomó al segundo algoritmo elegido por el usuario, por ejemplo: [0.0002, 0.005, 0.019]. Y luego las funciones “title()” es para el título, “xlabel()” es la etiqueta del eje x, “ylabel()” igual pero del eje Y, “legend()” es el recuadro de arriba a la izquierda que indica el color para cada línea, “grid()” hace ver la cuadrícula, “tight_layout()” es una funcion que ajusta automaticamente los espacios entre etiquetas y titulos, y “show()” muestra el gráfico en pantalla.

```
def graficar():
    #grafica las líneas del grafico↓
    plt.plot(tamaños, tiempos_eleccion1, label=f"{eleccion1}", color="red", marker="o")
    plt.plot(tamaños, tiempos_eleccion2, label=f"{eleccion2}", color="blue", marker="s")
    # Etiquetas
    plt.title("Comparación de Algoritmos de Ordenamiento")
    plt.xlabel("Cantidad de elementos")
    plt.ylabel("Tiempo de ejecución (segundos)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Siguiendo con el código, en el siguiente bloque se crean las listas de 100, 500, 1.000, 5.000 y 10.000 elementos aleatorios utilizando la librería Random que habíamos importado. Básicamente con bucles for se itera X número de veces generando un número aleatorio entre 1 y 10.000 en cada iteración, generando así la lista con X número de elementos aleatorios.

```
#Creacion de listas con 100, 500, 1.000, 5.000 y 10.000 elementos
lista100 = [random.randint(1,10000) for i in range(100)]
lista500 = [random.randint(1,10000) for i in range(500)]
lista1000 = [random.randint(1,10000) for i in range(1000)]
lista5000 = [random.randint(1,10000) for i in range(5000)]
lista10000 = [random.randint(1,10000) for i in range(10000)]
```

Luego se ejecutan dos “Input()” para hacer al usuario elegir entre los algoritmos de ordenamiento y se muestra el mensaje en pantalla. Los números que se introduzcan serán guardados en la variables “eleccion1” y “eleccion2”

```
#interfaz de usuario
eleccion1 = int(input("Ingrese el algoritmo que desea comparar: \n1: Bubble Sort \n2: Insertion Sort \n3: Selection Sort \n4: Quicksort \n5: Sorted(): \n"))
eleccion2 = int(input("Ingrese el segundo algoritmo que desea comparar: \n1: Bubble Sort \n2: Insertion Sort \n3: Selection Sort \n4: Quicksort \n5: Sorted(): \n"))
```

Seguidamente, se verifica lo que el usuario ingresó en los inputs, si su respuesta fue 1, significa que su elección fue “Bubble Sort” (sirve para darle nombre luego al gráfico), se reemplaza en la variable “eleccion1” de antes y se le asigna a la variable “funcion” la función correspondiente a esa elección, es decir, si el usuario eligió por ejemplo Bubble Sort en la primera ocasión, el valor de la variable “funcion1” será “bubble_sort”, que es como se llama la función que contiene el algoritmo. Luego veremos que se pasa dicho valor a la función “medidor_de_tiempos()” que vimos anteriormente para que para que ésta ejecute el algoritmo y mida su eficiencia.

```
if eleccion1 == 1:
    eleccion1 = "Bubble Sort"
    funcion1 = bubble_sort
elif eleccion1 == 2:
    eleccion1 = "Insertion Sort"
    funcion1 = insertion_sort
elif eleccion1 == 3:
    eleccion1 = "Selection Sort"
    funcion1 = selection_sort
elif eleccion1 == 4:
    eleccion1 = "Quicksort"
    funcion1 = quicksort
elif eleccion1 == 5:
    eleccion1 = "Sorted"
    funcion1 = sorted
```

Luego, hacemos lo mismo pero para el segundo algoritmo elegido por el usuario. Esta vez las variables son “eleccion2” y “funcion2”.

```
if eleccion2 == 1:
    eleccion2 = "Bubble Sort"
    funcion2 = bubble_sort
elif eleccion2 == 2:
    eleccion2 = "Insertion Sort"
    funcion2 = insertion_sort
elif eleccion2 == 3:
    eleccion2 = "Selection Sort"
    funcion2 = selection_sort
elif eleccion2 == 4:
    eleccion2 = "Quicksort"
    funcion2 = quicksort
elif eleccion2 == 5:
    eleccion2 = "Sorted"
    funcion2 = sorted
else:
    print("Error: valor ingresado no correspondiente. Finalizando programa...")
```

A continuación, se crean las variables “resultadoE1_x” y “resultadoE2_x” (Resultado de elección 1_x). Se van almacenando en estas variables los tiempos de ejecución de cada algoritmo, con la función “medidor_de_tiempos()”, a la cual se le pasa “funcion1” que contiene la función correspondiente al primer algoritmo elegido, así como “funcion2” que contiene la función del segundo algoritmo elegido por el usuario.

```
resultadoE1_1 = medidor_de_tiempos(funcion1, lista100)
resultadoE1_2 = medidor_de_tiempos(funcion1, lista500)
resultadoE1_3 = medidor_de_tiempos(funcion1, lista1000)
resultadoE1_4 = medidor_de_tiempos(funcion1, lista5000)
resultadoE1_5 = medidor_de_tiempos(funcion1, lista10000)

resultadoE2_1 = medidor_de_tiempos(funcion2, lista100)
resultadoE2_2 = medidor_de_tiempos(funcion2, lista500)
resultadoE2_3 = medidor_de_tiempos(funcion2, lista1000)
resultadoE2_4 = medidor_de_tiempos(funcion2, lista5000)
resultadoE2_5 = medidor_de_tiempos(funcion2, lista10000)
```

Luego, se le pregunta mediante otro “input()” al usuario si quiere mostrar los resultados de las listas pequeñas o si quiere mostrar los de las listas grandes también:

```
eleccion3 = int(input("¿En qué listas quiere probar los algoritmos? \n1: Listas pequeñas (de 100 a 1000 elementos)\n2: Listas grandes y pequeñas (de 100 a 10.000 elementos)\n"))
```

Y finalmente, se valida la opción del usuario, si es 1 significa que el usuario quiere ver los resultados de los algoritmos con listas pequeñas por lo que se hace un eje X con los tamaños de listas [100, 500, 1000], se cargan los resultados de los algoritmos elegidos y se llama a la función “graficar()” que utiliza estos datos para crear el gráfico y mostrarlo en pantalla. Así también si el usuario elige 2, en vez de hacerse un gráfico tres valores de X, se hace uno con 5 valores y se cargan los resultados de las ejecuciones de los algoritmos elegidos, al igual que la otra opción. En ambas opciones se imprimen los valores en segundos que utiliza el gráfico. Si se elige otra opción que no sea 1 o 2, se lanza el error: “Error: valor ingresado no correspondiente. Finalizando el programa...”.

```
#elecciones
if eleccion3 == 1:
    tamaños = [100, 500, 1000]
    tiempos_eleccion1 = [resultadoE1_1, resultadoE1_2, resultadoE1_3]
    tiempos_eleccion2 = [resultadoE2_1, resultadoE2_2, resultadoE2_3]
    graficar()
    print(f"Resultados de {eleccion1}: {resultadoE1_1, resultadoE1_2, resultadoE1_3}\nResultados de {eleccion2}: {resultadoE2_1, resultadoE2_2, resultadoE2_3}")
elif eleccion3 == 2:
    tamaños = [100, 500, 1000, 5000, 10000]
    tiempos_eleccion1 = [resultadoE1_1, resultadoE1_2, resultadoE1_3, resultadoE1_4, resultadoE1_5]
    tiempos_eleccion2 = [resultadoE2_1, resultadoE2_2, resultadoE2_3, resultadoE2_4, resultadoE2_5]
    graficar()
    print(f"Resultados de {eleccion1}: {resultadoE1_1, resultadoE1_2, resultadoE1_3, resultadoE1_4, resultadoE1_5}\nResultados de {eleccion2}: {resultadoE2_1, resultadoE2_2, resultadoE2_3, resultadoE2_4, resultadoE2_5}")
else:
    print("Error: valor ingresado no correspondiente. Finalizando programa...")
```

4. Caso Práctico

Para llevar a cabo la comparación entre distintos algoritmos de ordenamiento, se siguieron los siguientes pasos:

Selección de algoritmos:

Se eligieron cinco algoritmos de ordenamiento representativos:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quicksort
- La función integrada sorted() de Python (que utiliza Timsort).

Generación de datos:

Se generaron listas aleatorias con tamaños crecientes: 100, 500, 1000, 5000 y 10.000 elementos. Los valores fueron enteros aleatorios entre 1 y 10.000, utilizando la biblioteca random.

Medición de tiempos:

Para medir el tiempo de ejecución de cada algoritmo, se utilizó la función `default_timer()` del módulo **timeit**, registrando el tiempo antes y después de la ejecución de cada ordenamiento. Esta medición se realizó una vez por lista y por algoritmo.

Comparación y visualización:

Los resultados obtenidos (tiempos en segundos) se graficaron utilizando `matplotlib.pyplot`, mostrando cómo varía el rendimiento de cada algoritmo en función del tamaño de la lista a ordenar.

Se trazaron curvas para cada algoritmo con diferente color y marcador, y se incluyeron etiquetas, leyendas y formato visual claro mediante `tight_layout()` y `plt.grid()`.

Interfaz de usuario:

Se permitió al usuario elegir dos algoritmos a comparar mediante un menú por consola, de forma que el programa fuera interactivo y flexible.

5. Resultados obtenidos

- Se logró llevar a cabo un programa en python para la comparación de dos algoritmos de ordenamiento.
- Se evaluó el desempeño de **cinco algoritmos de ordenamiento**:
 - Bubble Sort
 - Insertion Sort
 - Selection Sort
 - Quicksort
 - `sorted()` (función interna de Python basada en Timsort)

Cada algoritmo fue probado con listas de tamaño **100, 500, 1.000, 5.000 y 10.000**, con valores generados aleatoriamente en cada ejecución. El objetivo fue analizar el **tiempo de ejecución** y cómo este crece con el tamaño de la entrada.

Tabla de tiempos de ejecución (en segundos):

Tamaño de lista	Bubble Sort	Insertion Sort	Selection Sort	Quicksort	sorted()
100	0.0003	0.0001	0.0001	0.000008	0.00000068
500	0.0008	0.0026	0.0039	0.0004	0.0000296
1000	0.0285	0.0107	0.0132	0.0008	0.0000679
5000	0.7091	0.2791	0.2920	0.0050	0.0004061
10000	2.8499	1.1519	1.1198	0.0107	0.0008877

Análisis de resultados:

- **Bubble Sort, Selection Sort e Insertion Sort** muestran un crecimiento muy marcado en el tiempo de ejecución, especialmente a partir de 1000 elementos. Esto coincide con su **complejidad $O(n^2)$** .
- **Quicksort** presenta una mejora significativa en listas grandes, aunque puede verse afectado por la elección del pivote si no está optimizado.
- La función **sorted()** de Python fue consistentemente la más rápida, gracias a que utiliza **Timsort**, un algoritmo híbrido altamente eficiente para listas parcialmente ordenadas y aleatorias.

6. Video presentación

- Link video presentación en Youtube: <https://youtu.be/hUDQ2KBJr4>