“Algoritmos de Búsqueda y Ordenamiento”

- **Alumnos**: Agustín Rivarola - agustin.rivarola@tupad.utn.edu.ar

  Esteban Rivarola - esteban.rivarola@tupad.utn.edu.ar

- **Materia**: Programación I
- **Profesor/a**: Prof. Nicolás Quirós
- **Fecha de Entrega**: 09/06/2025

Índice

---

1. Introducción

Los algoritmos de búsqueda y ordenamiento son herramientas clave en una estructura de datos, ya que permiten encontrar elementos específicos de manera eficiente y organizarlos siguiendo un criterio determinado.

En el siguiente trabajo se busca poder aplicar y explicar el funcionamiento en una situación concreta, lo que permite consolidar la comprensión de las funciones y comenzar a explorar los conceptos de eficiencia y rendimiento de un código.

Se eligió este tema para investigar ya que permite aplicar de forma práctica todas las unidades previamente exploradas, tales como: Estructuras condicionales, bucles, estructuras repetitivas, definición de funciones y listas, todas necesarias para aplicar en funciones de búsqueda y ordenamiento.

2. Marco Teórico

Búsqueda Lineal

La lógica del algoritmo de búsqueda lineal es sencilla, examina cada elemento de la lista uno por uno, comenzando desde el primer elemento, hasta que encuentra el elemento buscado o llega al final de la lista. El algoritmo recorre toda la lista sin importar si está ordenada o no.

- Ejemplo de cómo trabaja el algoritmo:

Objetivo para buscar: 1

Lista [3, 5, 1, 7, 9, 0, 2]

| 3 | 5 | 1 | 7 | 9 | 0 | 2 |
^

- El elemento en la posición 0 es 3 = no es 1

| 3 | 5 | 1 | 7 | 9 | 0 | 2 |
^

- El elemento en la posición 1 es 5 = no es 1

| 3 | 5 | 1 | 7 | 9 | 0 | 2 |
^

- El elemento en la posición 2 es 1 = ¡Sí, es 1!

- Implementación:

Es sencilla y fácil de implementar, pero puede ser ineficiente para grandes listas.

- Eficiencia:

Tiene una complejidad temporal de O(n), lo que significa que en el peor caso (cuando el elemento está al final o no está) podría necesitar comparar el elemento objetivo con todos los elementos de la lista.

- Uso:

Es adecuada para listas pequeñas o desordenadas, o cuando se realiza una única búsqueda en una lista.

Búsqueda Binaria

Los algoritmos de búsqueda binaria son rápidos y eficaces en comparación con los algoritmos de búsqueda lineal. Lo más importante de la búsqueda binaria es que sólo funciona con listas de elementos ordenadas. Si la lista no está ordenada, el algoritmo primero ordena los elementos utilizando el algoritmo de ordenación y luego la función de búsqueda binaria para encontrar la salida deseada. La búsqueda Binaria Divide la lista en dos partes y busca en la mitad correspondiente, reduciendo el tamaño del problema con cada paso

- Ejemplo de cómo trabaja el algoritmo:

Objetivo para buscar: 5

Lista ordenada: [0, 1, 2, 3, 5, 7, 9]

- Primer paso, La búsqueda binaria utiliza los punteros low (bajo), high (alto) y mid (medio) para reducir el espacio de búsqueda.

| 0 | 1 | 2 | 3 | 5 | 7 | 9 |
^ ^ ^
low mid high

- low = 0 (primer objeto)
- high = 6 (último objeto)
- mid = (0+6) // 2 = 3

- Comparamos el elemento[mid] (que es [3]=3) con el objetivo 5.
- 3 < 5, lo que significa que el objetivo está en la mitad superior (a la derecha de mid).
- Actualizamos low = mid + 1.

- Segundo paso, actualización de low
- low = 4
- high = 6
- mid = (4 + 6) // 2 = 5

| 0 | 1 | 2 | 3 | 5 | 7 | 9
^ ^ ^
low mid high

- Comparamos el objeto[mid] (que es [5]=7) con el objetivo 5.
- 7 > 5, lo que significa que el objetivo está en la mitad inferior (a la izquierda de mid).
- Actualizamos high = mid - 1.

- Tercer paso, actualización de high
- low = 4

- high = 4
- mid = (4 + 4) // 2 = 4

| 0 | 1 | 2 | 3 | 5 | 7 | 9 |
^
bajo/medio/alto

- Comparamos el objeto[mid] (que es [4]=5) con el objetivo 5.
- 5 == 5. ¡Sí, es 5!
- Resultado**:** El objetivo 5 fue encontrado en el **índice 4**

- Requisitos:

La lista debe estar ordenada previamente.

- Implementación:

Es más compleja que la búsqueda lineal, pero más eficiente para grandes listas.

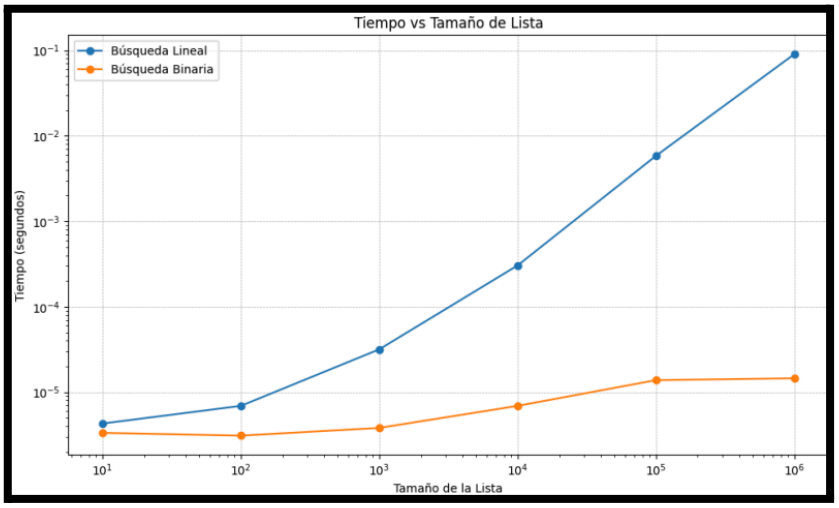
- Eficiencia:

Tiene una complejidad temporal de O(log n), lo que la hace mucho más rápida que la búsqueda lineal para listas grandes.

- Uso:

Ideal para listas grandes y ordenadas, donde la eficiencia es crucial.

Comparación de eficiencia:



La búsqueda binaria es significativamente más eficiente que la búsqueda lineal a medida que el tamaño de la lista aumenta, mostrando un menor tiempo de ejecución, especialmente para listas grandes.

Características	Búsqueda Lineal	Búsqueda Binaria
Requisito de datos	Sin orden específico (puede estar desordenada)	**Debe estar ordenada**
Eficiencia	Menos eficiente para grandes conjuntos de datos (O(n))	Altamente eficiente para grandes conjuntos de datos (O(log n))
Enfoque	Revisa cada elemento secuencialmente	Divide el espacio de búsqueda por la mitad repetidamente
Adecuado para	Listas pequeñas, desordenadas; encontrar todas las ocurrencias	Listas grandes, ordenadas; encontrar una sola ocurrencia
Complejidad	Sencillo de implementar	Ligeramente más complejo de implementar

Ordenamiento

Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente, ordenar los datos de acuerdo con un criterio, como de menor a mayor o alfabéticamente.. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Algunos de los beneficios de utilizar algoritmos de ordenamiento incluyen:

- Búsqueda más eficiente:

Una vez que los datos están ordenados, es mucho más fácil buscar un elemento específico. Esto se debe a que se puede utilizar la búsqueda binaria, que es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal.

- Análisis de datos más fácil:

Los datos ordenados pueden ser analizados más fácilmente para identificar patrones y tendencias. Por ejemplo, si se tienen datos sobre las ventas de una empresa, se pueden ordenar por producto, región o fecha para ver qué productos se venden mejor, en qué regiones se venden más productos o cómo cambian las ventas con el tiempo.

- Operaciones más rápidas:

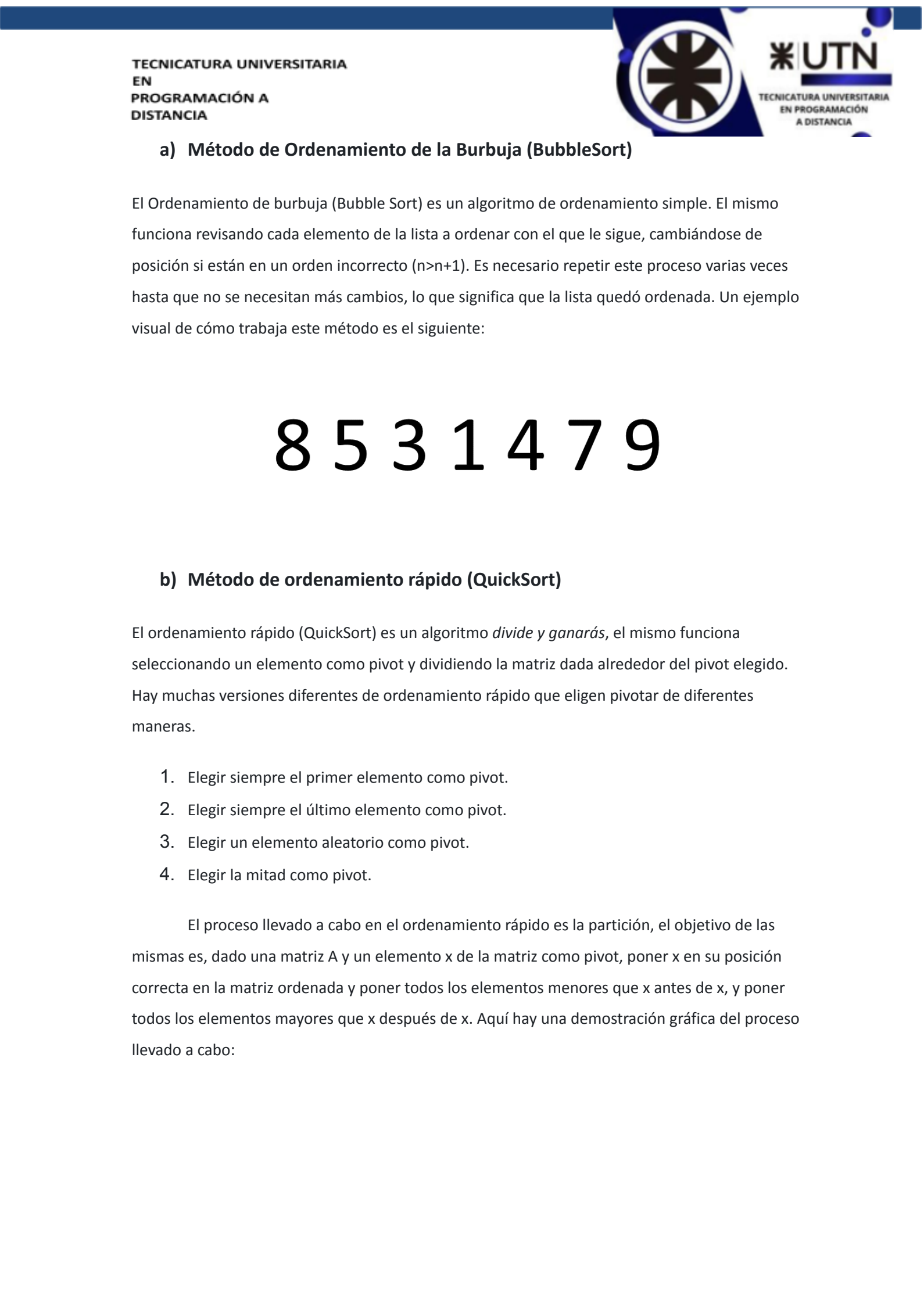
Muchas operaciones, como fusionar dos conjuntos de datos o eliminar elementos duplicados, se pueden realizar de manera más rápida y eficiente en datos ordenados.

Métodos de Ordenamiento

Los métodos de ordenamiento son algoritmos que realizan la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento. El ordenamiento se efectúa con base en el valor de algún campo en un grupo de datos. El ordenamiento puede estar dado de forma iterativa o recursiva según la naturaleza y forma de ejecución del mismo.
Estos son dos de los ejemplos más comunes de métodos de ordenamiento en Python:

a) Método de Ordenamiento de la Burbuja (BubbleSort)

El Ordenamiento de burbuja (Bubble Sort) es un algoritmo de ordenamiento simple. El mismo funciona revisando cada elemento de la lista a ordenar con el que le sigue, cambiándose de posición si están en un orden incorrecto (n>n+1). Es necesario repetir este proceso varias veces hasta que no se necesitan más cambios, lo que significa que la lista quedó ordenada. Un ejemplo visual de cómo trabaja este método es el siguiente:



b) Método de ordenamiento rápido (QuickSort)

El ordenamiento rápido (QuickSort) es un algoritmo *divide y ganarás*, el mismo funciona seleccionando un elemento como pivot y dividiendo la matriz dada alrededor del pivot elegido. Hay muchas versiones diferentes de ordenamiento rápido que eligen pivotar de diferentes maneras.

1. Elegir siempre el primer elemento como pivot.
2. Elegir siempre el último elemento como pivot.
3. Elegir un elemento aleatorio como pivot.
4. Elegir la mitad como pivot.

El proceso llevado a cabo en el ordenamiento rápido es la partición, el objetivo de las mismas es, dado una matriz A y un elemento x de la matriz como pivot, poner x en su posición correcta en la matriz ordenada y poner todos los elementos menores que x antes de x, y poner todos los elementos mayores que x después de x. Aquí hay una demostración gráfica del proceso llevado a cabo:

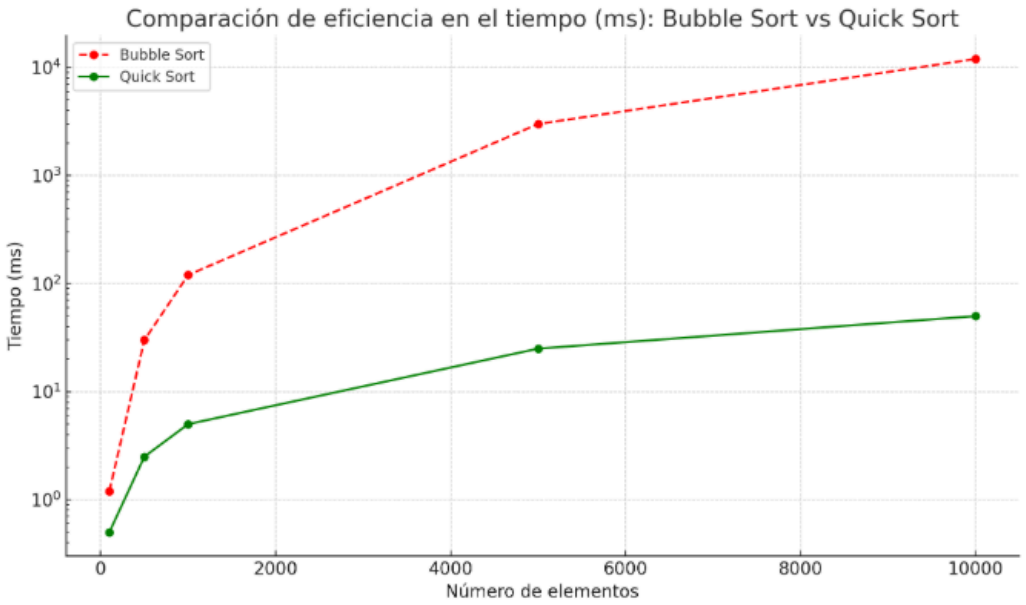
6 5 3 1 8 7 2 4

Comparación de eficiencia entre ambos métodos

Puntos Clave a Considerar:

- **Rendimiento:** La diferencia más significativa es el rendimiento. QuickSort es considerablemente más rápido que Bubble Sort para la mayoría de los casos, especialmente con grandes conjuntos de datos.
- **Complejidad:** La complejidad de Bubble Sort lo hace impráctico para la mayoría de las aplicaciones reales.
- **Estabilidad:** Si el orden relativo de elementos idénticos es importante, Bubble Sort es una mejor opción. Sin embargo, en la mayoría de los escenarios, esto no es una preocupación.
- **Implementación:** Bubble Sort es mucho más sencillo de entender e implementar, lo que lo hace útil para propósitos educativos o para arrays extremadamente pequeños.

En resumen, mientras que Bubble Sort es fácil de entender, Quick Sort es el algoritmo preferido para la ordenación de datos en la mayoría de las aplicaciones prácticas debido a su eficiencia superior.



---

2. Caso Práctico

<u>Ordenamiento Bubble Sort y Quick Sort</u>

Para el siguiente caso práctico de ordenamiento se eligió el método Bubble Sort (o Burbuja) y Quick Sort (o Rápido) para comparar la eficiencia y rapidez entre ambos métodos. Se creó una lista de números aleatorios y se ejecutó, se midieron los tiempos de procesamiento de la CPU con la función time.process_time() y se corroboró que el método Quick Sort, en general, es más eficiente que Bubble Sort, especialmente para conjuntos de datos grandes.

```
#importamos las librerías necesarias
import random #importamos la librería random para generar números aleatorios
import time #importamos la librería time para medir el tiempo de ejecución
```

```
lista = random.sample(range(1, 10000), 9990)# Generamos una lista de 9900 números aleatorios entre 1 y 10000
```

```
#Método de ordenamiento burbuja
def burbuja(lista):# definición de la función burbuja
    n = len(lista)# obtenemos la longitud de la lista
    for i in range(n):# iteramos sobre la lista n veces
        for j in range(0, n-i-1):# iteramos sobre la lista desde el inicio hasta el final menos i
            # comparamos los elementos adyacentes y los intercambiamos si están en el orden incorrecto.
            if lista[j] > lista[j+1]:#
                lista[j], lista[j+1] = lista[j+1], lista[j] #
    return lista    #retornamos la lista ordenada
```

```python
#Método de ordenamiento quicksort

def quicksort(lista): #definición de la función quicksort
    if len(lista) <= 1: #caso base: si la lista tiene 0 o 1 elementos, ya está ordenada
        return lista
    else: # caso recursivo: se elige un pivote y se divide la lista en tres partes
        # elementos menores que el pivote, elementos iguales al pivote y elementos mayores que el pivote
        # luego se ordenan recursivamente las dos partes y se combinan
        pivot = lista[len(lista) // 2]
        izquierda = [x for x in lista if x < pivot]
        medio = [x for x in lista if x == pivot]
        derecha = [x for x in lista if x > pivot]
        return quicksort(izquierda) + medio + quicksort(derecha)# se combinan las partes ordenadas y se devuelve la lista ordenada
```

```python
# Ejemplo de uso buble sort
# Se mide el tiempo de ejecución del método burbuja
inicio_tiempo_burbuja = time.process_time()# inicia el conteo del tiempo de ejecución
lista_ordenada_burbuja = burbuja(lista)# ordena la lista usando el método burbuja
fin_tiempo_burbuja = time.process_time()# finaliza el conteo del tiempo de ejecución
# mostramos por pantalla el tiempo de ejecución del método burbuja
print(f"tiempo de ordenamiento burbuja: {fin_tiempo_burbuja - inicio_tiempo_burbuja} segundos")
```

```python
# Ejemplo de uso quicksort
#Se mide el tiempo de ejecución del método quicksort
inicio_tiempo = time.process_time()# inicia el conteo del tiempo de ejecución
lista_ordenada = quicksort(lista)# ordena la lista usando el método quicksort
fin_tiempo = time.process_time()# finaliza el conteo del tiempo de ejecución
#mostramos por pantalla el tiempo de ejecución del método quicksort
print("Tiempo de ordenamiento quicksort:", fin_tiempo - inicio_tiempo, "segundos")
```

```
Tiempo de ordenamiento quicksort: 0.015625 segundos
tiempo de ordenamiento burbuja: 4.390625 segundos
```

---

3. Metodología Utilizada

- Investigación de los métodos de ordenamiento y búsqueda. Las fuentes fueron los videos de la cátedra, los apuntes, repositorios en Github y trabajos particulares sobre el tema.
- División del trabajo, Agustín se encargó de los métodos de búsqueda y Esteban de los métodos de ordenamiento (Tanto en el desarrollo del marco teórico como en el caso práctico en Python)
- Agustín trabajó para compilar los audios, diapositivas y videos para la presentación final.

---

4. Resultados Obtenidos

Con el método de **ordenamiento**, se comparó el tiempo de ejecución de la CPU entre el método de Bubble Sort y Quick Sort y, de una lista de 10000 elementos únicos ordenados aleatoriamente, el resultado fue, aproximadamente:

```
Tiempo de ordenamiento quicksort: 0.015625 segundos
tiempo de ordenamiento burbuja: 4.25 segundos
```

Lo que se corroboró que, en listas grandes el método Quick Sort es más eficiente y rápido.

- Link del repositorio GitHub de la documentación: https://github.com/Tevan88/TP-integrador-1-Programaci-n

---

5. Conclusiones

En el presente trabajo exploramos y aprendimos acerca de los métodos de ordenamiento, tanto en su sintaxis como funciones en Python, en la eficiencia y rapidez entre el método Bubble Sort y Quick Sort. También sobre el módulo **time** para medir tiempos de ejecución y prácticas sobre recursividades.

Pusimos en práctica todo lo aprendido en el año: estructuras secuenciales, estructuras condicionales, estructuras repetitivas, listas, bibliotecas y funciones. Sentimos que este trabajo nos acercó un poco más a estructuras más complejas donde, gracias a lo aprendido anteriormente, pudimos entender de una manera más profunda estos contenidos, cómo crearlos y poder interpretarlos.

En un futuro se podría trabajar con algún caso testigo aplicable a la vida laboral y así implementar los algoritmos de búsqueda y ordenamiento teniendo una base de conocimiento en la eficiencia de cada algoritmo.

---

6. Bibliografía

- Búsqueda y Ordenamiento en programación. (2025). Tecnicatura Universitaria en Programación. Cátedra Programación 1. *Teoría sobre Búsqueda y Ordenamiento.*

- Análisis comparativo de algoritmos de ordenamiento. (2018). *Sergio Alexander* https://www.pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/

- Métodos de ordenamiento. (2020). *Github de gbaudino*. https://github.com/gbaudino/MetodosDeOrdenamiento?tab=readme-ov-file

---

7. Anexos

- Link video en Youtube: https://www.youtube.com/watch?v=9GNRqHxIu_E
- Link de GitHub: https://github.com/Tevan88/TP-integrador-1-Programaci-n


UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA