Trabajo Integrador – Algoritmos de Búsqueda y Ordenamiento.

Tecnicatura Universitaria en Programación

Año 2025

Alumnos:

- Gabriel Torres – jose.torres@tupad.utn.edu.ar
- Daiana Velasquez – daiana.velasquez@tupad.utn.edu.ar

Materia: Programación I

Profesor/a: Nicolas Quirós.

Tutor: Matias Santiago Torres.

Fecha de Entrega: 09/06/2025

Índice

Introducción

El tema de **algoritmos de búsqueda y ordenamientos** fue elegido debido a su carácter fundamental en el campo de la programación y la informática. Estas técnicas constituyen la base para el manejo y procesamiento eficiente de datos, siendo aplicables a una amplia variedad de problemas reales. Además, su implementación en Python permite demostrar de manera práctica cómo estos algoritmos impactan en el rendimiento de los programas.

En el ámbito de la programación y la informática, los algoritmos de búsqueda y ordenamiento representan fundamentos esenciales para el manejo eficiente de datos. Estas técnicas permiten **localizar información de forma rápida y organizar datos de manera eficiente**, facilitando el acceso, la visualización y el análisis. Python, como uno de los lenguajes de programación más populares y versátiles en la actualidad, ofrece múltiples herramientas y bibliotecas que simplifican la implementación de estos algoritmos.

El estudio de los algoritmos de búsqueda y ordenamiento en Python no solo es relevante desde un punto de vista académico, sino también práctico, ya que estos procesos son comunes en una gran variedad de aplicaciones, desde bases de datos hasta inteligencia artificial. Entender cómo funcionan estos algoritmos y cómo se implementan en Python permite desarrollar programas más rápidos, eficientes y escalables, logrando así una optimización de los tiempos de respuesta y de recursos cuando sea posible.

En este trabajo abordaremos los conceptos fundamentales de búsqueda y ordenamiento, explorando tanto algoritmos simples como búsqueda lineal y ordenamiento por burbuja, como también técnicas más complejas como búsqueda binaria y algoritmos de ordenamiento eficientes como Quicksort, con ejemplos aplicados en Python.

Marco teórico

Búsqueda

La búsqueda consiste en localizar un elemento en un conjunto de datos. Es una herramienta que se utiliza en una amplia variedad de aplicaciones de programación. Es también una operación fundamental en programación, se utiliza para encontrar un elemento específico dentro de un conjunto de datos.

Es importante porque se utiliza en una amplia variedad de aplicaciones, es una tarea común en muchas de ellas, como bases de datos, sistemas de archivos y algoritmos de inteligencia artificial. Al comprender los diferentes algoritmos de búsqueda y cómo utilizarlos, se puede mejorar el rendimiento y la eficiencia de los programas.

El tamaño de la lista tiene un impacto significativo en el tiempo que tardan los algoritmos de búsqueda en encontrar el objetivo. Cuanto más grande sea la lista, más tiempo tardará el algoritmo en encontrar el elemento deseado. Esto se debe a que el algoritmo debe comprobar cada elemento de la lista hasta encontrar el que busca.

Algoritmos de Búsqueda:

Los algoritmos de búsqueda son fundamentales en ciencias de la computación, ya que permiten localizar uno o más elementos dentro de una estructura de datos, como listas, tuplas o diccionarios. En Python, existen diversos métodos de búsqueda, desde técnicas simples hasta algoritmos más eficientes, dependiendo del tipo y la organización de los datos.

A continuación, se describen los principales algoritmos de búsqueda:

- **Búsqueda Lineal (Sequential Search):**

Itera sobre cada elemento de la lista hasta encontrar el objetivo o llegar al final. Es simple, pero puede ser lenta si hay muchos datos.

- **Búsqueda Binaria (Binary Search):**

Requiere que la lista esté ordenada, esto es fundamental, ya que si no está ordenada el resultado podría ser erróneo. Funciona dividiendo repetidamente la lista ordenada por la mitad, para determinar en qué segmento podría encontrarse el valor buscado.

COMPARACION ENTRE AMBOS METODOS DE BUSQUEDA

Criterio	Búsqueda Lineal	Búsqueda Binaria
Requisito de la lista	No requiere que los datos estén ordenados	Requiere que los datos estén previamente ordenados
Método	Recorre cada elemento uno por uno	Divide el arreglo a la mitad en cada paso
Complejidad Temporal (peor caso)	O(n)	O(log n)
Complejidad Espacial	O(1) (no requiere memoria adicional)	O(1) iterativa, O(log n) si es recursiva
Eficiencia en conjuntos pequeños o desordenados	Alta (simple y directa)	Ineficiente si el arreglo no está ordenado
Eficiencia en grandes volúmenes ordenados	Baja eficiencia	Muy eficiente

Facilidad de implementación	Muy simple	Moderadamente simple
Aplicaciones comunes	Búsqueda en listas no ordenadas, estructuras lineales (como arrays)	Búsqueda eficiente en bases de datos o estructuras ordenadas

La búsqueda lineal es útil cuando se trabaja con listas pequeñas o no ordenadas, debido a su simplicidad y flexibilidad. En cambio, la búsqueda binaria es mucho más rápida para conjuntos grandes **siempre que los datos estén ordenados previamente**, reduciendo significativamente el número de comparaciones necesarias.

Algoritmos de Ordenamiento:

Los algoritmos de ordenamiento (o "sorting algorithms") son procedimientos diseñados para reorganizar una colección de elementos (como números, cadenas o estructuras de datos) en un orden determinado, generalmente ascendente o descendente. Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Su aplicación es fundamental en la programación, ya que muchas operaciones posteriores —como búsquedas eficientes, estadísticas o visualización de datos— dependen de datos ordenados. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Python proporciona funciones incorporadas para ordenar listas (por ejemplo, list.sort() y sorted()) y también permite implementar algoritmos de búsqueda y ordenamiento personalizados. La elección del algoritmo depende de factores como el tamaño de la lista, si está ordenada y los requisitos de rendimiento.

En Python, existen múltiples formas de implementar algoritmos de ordenamiento. A continuación, se describen algunos de los más representativos:

- **Ordenamiento por Inserción (Insertion Sort):** Inserta cada elemento en su posición correcta dentro de la lista ordenada hasta el momento. Es eficiente para listas pequeñas o parcialmente ordenadas.
- **Ordenamiento por Selección (Selection Sort):** Encuentra el elemento más pequeño (o más grande) en la lista y lo mueve a la posición correcta. Es más eficiente que el Bubble Sort, pero sigue siendo lento para listas grandes.
- **Ordenamiento por Burbuja (Bubble Sort):** Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Es fácil de entender, pero no muy eficiente para listas grandes.

- **Ordenamiento por Burbuja Mejorado (Bubble Sort Mejorado):** Consiste en el mismo algoritmo (bubble sort) pero con unas ligeras modificaciones para evitar que sus dos ciclos sigan recorriendo la lista cuando los elementos ya están ordenados. Esto nos permite una optimización del tiempo en algunos casos.
- **Ordenamiento Rápido (Quick Sort):** Divide la lista en sublistas y recursivamente las ordena. Es mucho más rápido que el Bubble Sort en la mayoría de los casos.

En Python, los algoritmos de búsqueda y ordenamiento son herramientas fundamentales para manipular y gestionar datos. Los algoritmos de búsqueda permiten encontrar elementos específicos dentro de una estructura de datos, mientras que los algoritmos de ordenamiento organizan los elementos en un orden específico.

Implementación de Algoritmos de búsquedas de datos en Python:

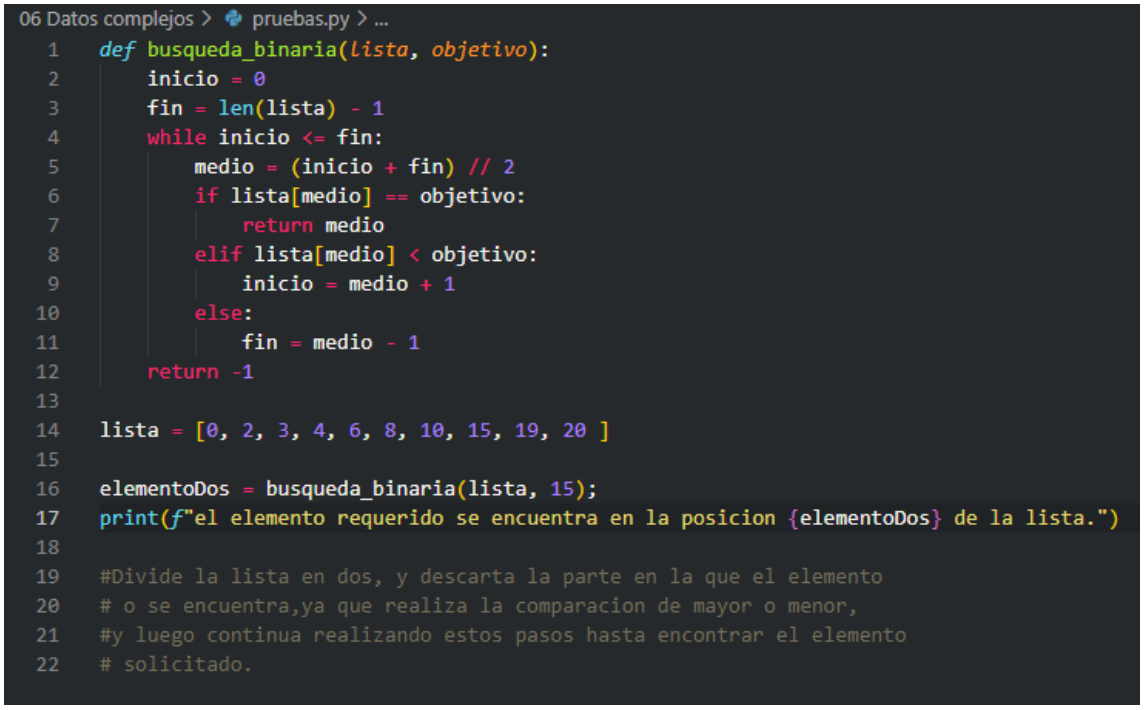
Algoritmos de Búsqueda lineal en Python.

```python
#  Busqueda Lineal
# Busca un elemento recorriendo toda la lista.
import time
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Definir la lista de ejemplo
lista = [9, 2, 4, 3, 6, 8, 1, 14, 12, 20]

# Medir tiempo de ejecución
t_inicio = time.time()
elemento = busqueda_lineal(lista, 8)
t_fin = time.time()

# Mostrar resultados
print(f"El elemento requerido se encuentra en la posición {elemento} de la lista.")
print(f"Tiempo de ejecución: {t_fin - t_inicio:.10f} segundos.")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Microsoft/Wind
El elemento requerido se encuentra en la posición 5 de la lista.
Tiempo de ejecución: 0.0000000000 segundos.
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1>
```

Acá podemos ver como al utilizar una función de búsqueda lineal, al solicitar el elemento “7”, que se encuentra en la posición “4”, la consola nos devuelve la ubicación del elemento y también el tiempo de demora en la respuesta.

Algoritmo de búsqueda Binaria en Python.



Acá podemos ver como al solicitar el elemento “15”, que se encuentra en la posición “7”, la consola nos devolvería la ubicación del elemento.

Algoritmos de Ordenamiento en Python.

```python
import time
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

# Definir la lista de ejemplo
lista = [0, 2, 3, 4, 6, 8, 10, 15, 19, 20]

# Medir tiempo de ejecución
t_inicio = time.time()
elementoDos = busqueda_binaria(lista, 15)
t_fin = time.time()

# Mostrar resultados
print(f"El elemento requerido se encuentra en la posición {elementoDos} de la lista.")
print(f"Tiempo de ejecución: {t_fin - t_inicio:.10f} segundos.")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Microsoft/WindowsApps/py
El elemento requerido se encuentra en la posición 7 de la lista.
Tiempo de ejecución: 0.0000000000 segundos.
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1>
```

Ordenamiento por Inserción (Insertion Sort):

```python
def insertion_sort(lista):

    # Recorremos desde el segundo elemento hasta el final
    for i in range(1, len(lista)):
        valor_actual = lista[i]
        j = i - 1

        # Movemos los elementos de lista[0..i-1] que sean mayores que valor_actual
        # una posicion adelante de su posicion actual.
        while j >= 0 and lista[j] > valor_actual:
            lista[j + 1] = lista[j]
            j -= 1

        # Insertamos el valor en la posicion correcta.
        lista[j + 1] = valor_actual

    return lista

# Ejemplo de uso
numeros = [5, 2, 9, 1, 8, 6]
ordenados = insertion_sort(numeros)
print("Lista ordenada:", ordenados)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Microsoft/Windo
Lista ordenada: [1, 2, 5, 6, 8, 9]
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1>
```

En este caso, para ordenar la lista por inserción, recorrimos los elementos de la lista, comienza desde el segundo elemento y compara hacia atrás. Si encuentra un valor mayor, lo desplaza hacia adelante. Luego inserta el valor actual en el lugar correcto. En la consola podemos ver la lista ordenada.

Ordenamiento por Selección (Selection Sort):

```python
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        # Suponemos que el minimo esta en la posicion i
        indice_minimo = i

        # Buscamos el menor elemento en el resto de la lista
        for j in range(i + 1, n):
            if lista[j] < lista[indice_minimo]:
                indice_minimo = j

        # Intercambiamos el elemento actual con el menor encontrado
        lista[i], lista[indice_minimo] = lista[indice_minimo], lista[i]

    return lista

# ejemplo de uso
numeros = [29, 10, 14, 37, 13]
ordenados = selection_sort(numeros)
print("Lista ordenada:", ordenados)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Micr
Lista ordenada: [10, 13, 14, 29, 37]
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1>
```

En este caso, para ordenar la lista por selección, el algoritmo primero busca y encuentra el elemento más pequeño del arreglo y lo coloca en la primera posición. En la consola podemos ver la lista ordenada.

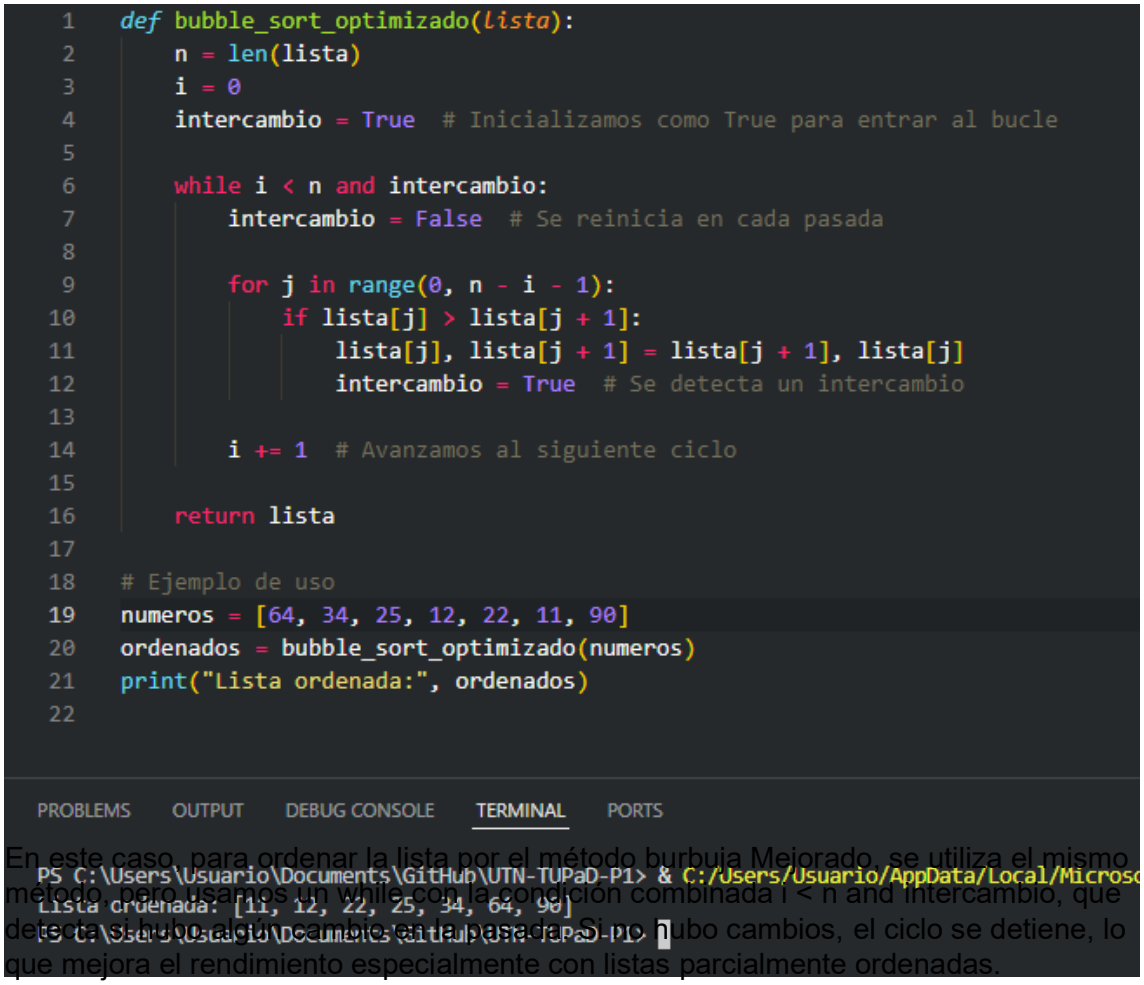
Ordenamiento por Burbuja (Bubble Sort):

```python
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
        #En cada pasada, los elementos mas grandes se desplazaran hacia el final.
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                #Se realiza un intercambio si estan en el orden incorrecto,
                # para ir acomodandolos.
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

#Ejemplo en consola
numeros = [64, 34, 25, 12, 22, 11, 90]
ordenados = bubble_sort(numeros)
print("Lista ordenada:", ordenados)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Microsoft/Windows
Lista ordenada: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1>
```

En este caso, para ordenar la lista por el método burbuja, compara pares de elementos adyacentes, es decir, que se encuentran de manera continua. Si están en el orden incorrecto, los intercambia (el menor siempre va primero). Repite el proceso varias veces, y en cada pasada, el mayor elemento se va al final, dejándonos así la lista ordenada.

Ordenamiento por Burbuja Mejorado (Bubble Sort Mejorado)



```python
def bubble_sort_optimizado(lista):
    n = len(lista)
    i = 0
    intercambio = True  # Inicializamos como True para entrar al bucle

    while i < n and intercambio:
        intercambio = False  # Se reinicia en cada pasada

        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                intercambio = True  # Se detecta un intercambio

        i += 1  # Avanzamos al siguiente ciclo

    return lista

# Ejemplo de uso
numeros = [64, 34, 25, 12, 22, 11, 90]
ordenados = bubble_sort_optimizado(numeros)
print("Lista ordenada:", ordenados)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Microso
Lista ordenada: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1>
```

En este caso, para ordenar la lista por el método burbuja Mejorado, se utiliza el mismo método, pero usamos un while con la condición combinada i < n and intercambio, que detecta si hubo algún cambio en la pasada. Si no hubo cambios, el ciclo se detiene, lo que mejora el rendimiento especialmente con listas parcialmente ordenadas.

Ordenamiento Rápido (Quick Sort)

```python
def quick_sort(lista):
    if len(lista) <= 1:
        return lista  # Caso base: lista vacia o con un solo elemento ya esta ordenada

    # Elegimos el pivote (puede ser el primer elemento, el ultimo, o aleatorio.)
    pivote = lista[0]

    # Partimos la lista en listas menores, iguales y mayores al pivote
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]

    # Aplicamos recursion y combinamos
    return quick_sort(menores) + [pivote] + quick_sort(mayores)

# Ejemplo de uso en consola
numeros = [10, 7, 8, 9, 1, 5]
ordenados = quick_sort(numeros)
print("Lista ordenada:", ordenados)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Usuario\Documents\GitHub\UTN-TUPaD-P1> & C:/Users/Usuario/AppData/Local/Microsoft/WindowsA
Lista ordenada: [1, 5, 7, 8, 9, 10]
```

En este caso, para ordenar la lista por el método de ordenamiento rapido, se selecciona un **pivote** (en este caso, el primer elemento). Luego se divide la lista en:

- Elementos **menores o iguales** al pivote.
- Elementos **mayores** al pivote.

Luego, se ordenan recursivamente ambos lados y se combinan.

3 -Caso práctico:

Odenamiento y Búsqueda en una biblioteca de Libros

1. Descripción del problema a resolver
Se desea simular el manejo de una colección de libros en una biblioteca. Cada libro está representado por su título, autor y año de publicación. El objetivo es:

-Generar una colección aleatoria de libros.
-Ordenarlos por año de publicación usando diferentes algoritmos.
-Buscar un libro específico mediante búsqueda binaria.

Esta simulación permite observar cómo impacta la elección del algoritmo de ordenamiento en el rendimiento del sistema, especialmente cuando se trata de trabajar con grandes volúmenes de datos.

```python
import random
import time

# 1. Generar biblioteca de libros aleatorios
titulos = ["Búsqueda", "Ordenamientos", "Códigos", "Redes", "Datos", "Python", "AI"]
autores = ["Ana", "Lautaro", "Marta", "Carlos", "Luna"]

biblioteca = []
for _ in range(1000):
    libro = {
        "titulo": random.choice(titulos),
        "autor": random.choice(autores),
        "año": random.randint(1980, 2025)
    }
    biblioteca.append(libro)
```

```python
# 2. Bubble Sort Mejorado (lento)
def bubble_sort(lista):
    arr = lista.copy()
    n = len(arr)
    for i in range(n):
        intercambiado = False
        for j in range(0, n - i - 1):
            if arr[j]["año"] > arr[j + 1]["año"]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                intercambiado = True
        if not intercambiado:
            break
    return arr

# 3. Quick Sort (rápido)
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if x["año"] <= pivote["año"]]
        mayores = [x for x in lista[1:] if x["año"] > pivote["año"]]
        return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

Se eligió Bubble Sort mejorado porque, aunque no es el algoritmo más eficiente para grandes volúmenes de datos, es sencillo de implementar y permite observar claramente cómo el rendimiento disminuye a medida que aumenta la cantidad de elementos.

Luego, se comparó con QuickSort, con el objetivo de analizar las diferencias en tiempos de ejecución entre algoritmos simples y algoritmos simples y algotirmos más eficientes.

```python

# 4. Búsqueda binaria por año
def busqueda_binaria(lista, año_objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio]["año"] == año_objetivo:
            return lista[medio]
        elif lista[medio]["año"] < año_objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return None

```

Se empleo la búsqueda binaria como método de localización, debido a su alta eficiencia de listas ordenadas, lo cual refuerza la importancia del proceso de ordenamiento previo.

```python

# 5. Ejecutar ordenamientos y medir tiempos
año_a_buscar = 2005

inicio = time.time()
orden_bubble = bubble_sort(biblioteca)
tiempo_bubble = time.time() - inicio
resultado_bubble = busqueda_binaria(orden_bubble, año_a_buscar)

inicio = time.time()
orden_quick = quick_sort(biblioteca)
tiempo_quick = time.time() - inicio
resultado_quick = busqueda_binaria(orden_quick, año_a_buscar)

# 6. Mostrar resultados
print("\n--- COMPARACIÓN DE ORDENAMIENTOS ---")
print(f"Bubble Sort tomó: {tiempo_bubble:.4f} segundos")
print(f"Quick Sort tomó : {tiempo_quick:.4f} segundos")
print(f"Diferencia      : {tiempo_bubble - tiempo_quick:.4f} segundos")

print("\n--- RESULTADOS DE BÚSQUEDA BINARIA ---")
print("Resultado en lista ordenada con Bubble Sort:", resultado_bubble)
print("Resultado en lista ordenada con Quick Sort:", resultado_quick)

```

Se realizaron pruebas con 1000 libros generados aleatoriamente. Se midió el tiempo que tarda cada algoritmo de ordenamiento y se verificó que la búsqueda binaria devuelva resultados correctos para años existentes. Los resultados demostraron que **QuickSort** fue significativamente más rápido que el **Bubble Sort mejorado**, cumpliendo así con los objetivos del análisis.

4 - Metodología Utilizada

Para abordar el análisis sobre **búsqueda y ordenamiento de datos**, se implementó una metodología estructurada que incluyó las siguientes etapas:

1. **Revisión Bibliográfica**:
   Se consultaron fuentes académicas, libros de texto y documentación técnica relacionadas con algoritmos clásicos y modernos de búsqueda y ordenamiento. Entre las referencias se incluyeron textos como *Introduction to Algorithms* (CLRS) y artículos de investigación sobre eficiencia algorítmica y estructuras de datos.
2. **Análisis Comparativo**:
   Se compararon distintos algoritmos de búsqueda y ordenamiento en términos de complejidad temporal y adaptabilidad a distintos tipos de datos. Algunos de los algoritmos evaluados fueron:
   a. Búsqueda lineal vs. búsqueda binaria.
   b. Ordenamiento por burbuja, selección e inserción (algoritmos simples).
   c. Ordenamiento rápido (QuickSort), ordenamiento por burbuja (BubbleSort), y ordenamiento por seleccion (selection sort).
3. **Estudio de Casos**:
   Se aplicaron algunos de estos algoritmos a conjuntos de datos para observar su comportamiento en escenarios diversos, tales como listas parcialmente ordenadas, datos con duplicados o grandes volúmenes de registros. También se evaluó la implementación en el lenguaje de programación Python.
4. **Simulación y Evaluación de Rendimiento**:
   Se realizaron pruebas prácticas y benchmarks para medir el tiempo de ejecución y la eficiencia en diferentes condiciones (por ejemplo, listas pequeñas vs. grandes, datos aleatorios vs. ordenados).

5 - Resultados Obtenidos

De los análisis realizados se obtuvieron los siguientes resultados:

Durante la evaluación de los distintos algoritmos de búsqueda y ordenamiento implementados en Python, se observaron diferencias significativas en cuanto a la eficiencia y el tiempo de ejecución según el tipo de algoritmo y la cantidad de elementos procesados.

En cuanto a los **algoritmos de búsqueda**, la **búsqueda lineal** demostró ser adecuada para listas pequeñas o no ordenadas, ya que recorre secuencialmente los elementos hasta encontrar el valor deseado. Sin embargo, al trabajar con listas de mayor tamaño, la **búsqueda binaria** resultó mucho más eficiente, siempre que los datos estuvieran previamente ordenados, debido a su menor complejidad temporal.

Respecto a los **algoritmos de ordenamiento**, los métodos simples como **BubbleSort Mejorado**, y **QuickSort** ofrecieron un rendimiento aceptable con conjuntos de datos pequeños. No obstante, su eficiencia disminuyó considerablemente a medida que crecía el número de elementos, debido a su complejidad O(n**2).

En contraste, el algoritmo **QuickSort** presentó mejores resultados en la mayoría de los casos, mostrando tiempos de ejecución notablemente menores en listas grandes gracias a su eficiencia promedio. Esto lo posiciona como una alternativa preferible en escenarios que demandan un rendimiento más alto.

En resumen, los resultados obtenidos confirmaron lo esperado teóricamente: los algoritmos con menor complejidad temporal fueron más eficientes en conjuntos de datos grandes, mientras que los algoritmos simples se desempeñaron adecuadamente en listas pequeñas o con estructuras particulares.

6- Conclusión

En conclusión, los algoritmos de búsqueda y ordenamiento constituyen pilares fundamentales en el desarrollo de software, proporcionando soluciones eficientes, escalables y precisas para la gestión de información, contribuyendo así a la creación de programas más rápidos, organizados y confiables. El estudio y la comparación de distintos algoritmos de búsqueda y ordenamiento permitieron comprender la importancia de seleccionar la técnica adecuada según el contexto y la naturaleza de los datos. Se evidenció que, si bien los algoritmos simples como la búsqueda lineal o el ordenamiento por burbuja pueden ser útiles en casos específicos o con volúmenes pequeños de información, su rendimiento se ve limitado en escenarios de mayor escala.

En cambio, algoritmos más eficientes como la búsqueda binaria y QuickSort demostraron ser más adecuados para conjuntos de datos grandes, ofreciendo tiempos de respuesta significativamente menores y mejor adaptabilidad. Esto resalta la necesidad de considerar tanto la complejidad algorítmica como las características del conjunto de datos antes de implementar una solución. Estas comparaciones son realizadas en el repositorio de GitHub.

En definitiva, el análisis permitió no solo evaluar el rendimiento de cada algoritmo, sino también reforzar conceptos fundamentales de eficiencia computacional y toma de decisiones en programación.

7 - Bibliografía

Introduction to Algorithms.
MIT Press, 3.ª edición, 2009.
Referencia clásica sobre algoritmos, incluyendo ordenamiento y búsqueda.

Algorithms.
Addison-Wesley Professional, 4.ª edición, 2011.

Learning Python.
O'Reilly Media, 5.ª edición, 2013.

Python Cookbook.
O'Reilly Media, 3.ª edición, 2013.

The Python Standard Library Documentation.
https://docs.python.org/3/library/
Fuente oficial para la documentación de funciones como sorted(), bisect, time, etc.

8 - Anexo

Repositorio con los códigos utilizados y desarrollados durante e l proyecto:
https://github.com/GabrielTorres25/UTN-TUPaD-P1/tree/main/06%20Datos%20complejos

Video de presentación del trabajo:

https://www.youtube.com/watch?v=7Kn9T-b7c5g