TRABAJO PRACTICO INTEGRADOR PROGRAMACIÓN I

ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO EN PHYTON

➢ **Estudiantes:**

Mauro Torres (lean.torres94@gmail.com)

Lorena Tropeano (ltropeano81@gmail.com)

➢ **Docentes:**

Profesor: Cinthia Rigoni

Tutor: Walter Pintos

➢ **Fecha de entrega:**

09/06/2025

INDICE

Introducción

El presente trabajo aborda el estudio de los algoritmos de ordenamiento y búsqueda implementados en el lenguaje de programación Python. Este tema fue seleccionado por su papel fundamental en el desarrollo de soluciones eficientes dentro de la informática, ya que muchas aplicaciones y programas requieren organizar datos y localizarlos rápidamente para funcionar de manera óptima.

Los algoritmos de ordenamiento y búsqueda son pilares de la programación, y su comprensión resulta esencial para cualquier estudiante o profesional del área. Permiten optimizar el tiempo de respuesta de los programas, reducir el uso de recursos y mejorar la experiencia del usuario. Además, son la base sobre la cual se construyen soluciones más complejas en inteligencia artificial, bases de datos, análisis de datos y desarrollo de software en general.

El objetivo de este trabajo es explorar dos enfoques distintos para resolver ese problema: un algoritmo tradicional implementado manualmente, y una solución moderna incorporada en el lenguaje Python: función sorted()

Marco Teórico

En programación, el tratamiento de datos requiere a menudo que la información esté ordenada. Ya sea para facilitar la búsqueda, mejorar la visualización o realizar cálculos estadísticos. El ordenamiento de listas y estructuras de datos es una de las tareas más frecuentes y fundamentales. Para resolver este problema, se utilizan los algoritmos de ordenamiento, que permiten reorganizar una colección de elementos siguiendo un criterio lógico (por ejemplo, de menor a mayor).

Existen diversas formas de ordenar datos en Python, que pueden clasificarse en funciones integradas del lenguaje, algoritmos manuales y ordenamientos avanzados personalizados.

Algoritmos de ordenamiento manuales

Con fines educativos, es común implementar algoritmos básicos como:

- Ordenamiento burbuja (Bubble Sort): compara e intercambia elementos adyacentes si están en el orden incorrecto. Es fácil de entender, pero lento para listas grandes.
- Ordenamiento por inserción (Insertion Sort): recorre la lista y va insertando cada elemento en su lugar adecuado. Es más eficiente que el burbuja en algunos casos, pero sigue siendo limitado para listas largas.
- Ordenamiento por selección (Selection Sort): encuentra el valor mínimo y lo coloca al principio, repitiendo el proceso hasta ordenar la lista. Tiene una lógica clara pero también es lento comparado con métodos modernos.

Ordenamientos personalizados

Python también permite ordenar por criterios más complejos utilizando funciones auxiliares o expresiones lambda. Por ejemplo, se puede ordenar una lista de nombres ignorando mayúsculas, o una lista de diccionarios por una propiedad interna como la edad o el promedio. Este tipo de ordenamientos se logra especificando un "criterio" que la función usa para comparar.

Esta capacidad convierte a Python en una herramienta muy flexible para trabajar con estructuras complejas y ordenar datos según las necesidades de cada caso.

Funciones integradas

Python ofrece herramientas muy potentes para ordenar listas sin necesidad de implementar algoritmos desde cero. Las dos más conocidas son sorted() y .sort(). Ambas permiten ordenar de forma ascendente o descendente, y son rápidas y eficientes. La diferencia principal es que sorted() devuelve una nueva lista ordenada, mientras que .sort() modifica la lista original.

Estas funciones utilizan internamente un algoritmo llamado Timsort, que combina las ventajas de Merge Sort e Insertion Sort. Esto les da un comportamiento muy eficiente incluso en listas grandes.

Importancia del análisis de algoritmos

Comprender cómo funcionan los distintos algoritmos permite al programador elegir la mejor opción según el contexto. En listas pequeñas, las diferencias pueden no ser evidentes, pero en listas grandes, elegir un mal algoritmo puede implicar pérdidas significativas de rendimiento. Por eso, en este trabajo se analiza y compara el comportamiento del ordenamiento burbuja y de la función sorted() para mostrar sus ventajas y limitaciones.

¿Cuál elegir?

Método	Cuándo usarlo
sorted()	Siempre que necesites una nueva lista ordenada
.sort()	Cuando querés modificar la lista original
Bubble/Insertion/Selection	Para aprender lógica de ordenamiento y en listas cortas
key o lambda	Para ordenar por propiedades o criterios personalizados

Caso Práctico

Para el caso práctico, pensamos en una escuela que tiene una lista con las notas finales de 500 alumnos que se deben ordenar de mayor a menor para premiar a los mejores promedios.

La idea fue comparar los códigos midiendo el tiempo de proceso de ambos para obtener ventajas y desventajas entre ellos.

Para el desarrollo del código en primer lugar se importan las librerías import y random, y luego se genera una lista de 500 notas aleatorias con dos decimales:

```
import random
import time

# Generar una lista de 500 notas aleatorias entre 0 y 10 con decimales
notas = [round(random.uniform(0, 10), 2) for _ in range(500)]
```

Luego se desarrolló el código Bubble Sort: definiendo una función a la que le pusimos ese nombre y recibe de parámetro una lista que tiene que ordenar.

```
# 1. Bubble Sort
def bubble_sort_desc(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] < lista[j + 1]:  # Para orden descendente
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

n: Calcula cuántos elementos tiene la lista. Se necesita saber la longitud para controlar los bucles.

for i in range(n): Este bucle controla cuántas vueltas se hacen sobre la lista. Se repite n veces, lo que garantiza que todos los elementos queden en orden.

for j in range(0, n - i - 1): Este segundo bucle recorre la lista comparando elementos adyacentes, pero se acorta en cada pasada porque al final de cada iteración, el elemento más grande ya se colocó en su lugar correcto.

if lista[j] < lista[j + 1]: Esta es la condición clave para ordenar de mayor a menor. Compara el elemento actual (lista[j]) con el siguiente (lista[j + 1]). Si el actual es menor que el siguiente, están en el orden incorrecto para una lista descendente.

lista[j], lista[j + 1] = lista[j + 1], lista[j]: Si están en el orden incorrecto**,** los intercambia de lugar. Así, el número más grande se coloca en el principio de la lista.

Se crean dos copias iguales de la misma lista para que ambos métodos trabajen igual

```
# Copias de la lista para que ambos métodos trabajen igual
notas_burbuja = notas.copy()
notas_sorted = notas.copy()
```

Luego se toman los tiempos y se llaman ambas funciones:

```python
# Tiempo con Bubble Sort
inicio_burbuja = time.time()
bubble_sort_desc(notas_burbuja)
fin_burbuja = time.time()
```

```python
# Tiempo con sorted()
inicio_sorted = time.time()
notas_sorted = sorted(notas_sorted, reverse=True)
fin_sorted = time.time()
```

Luego se muestran por pantalla ambos resultados:

```python
# Mostrar resultados
print("Top 10 notas ordenadas (Bubble Sort):", notas_burbuja[:10])
print("Top 10 notas ordenadas (sorted()):", notas_sorted[:10])
print(f"Tiempo Bubble Sort: {fin_burbuja - inicio_burbuja:.6f} segundos")
print(f"Tiempo sorted(): {fin_sorted - inicio_sorted:.6f} segundos")
```

Luego de ejecutarlo nos imprime:

```
Top 10 notas ordenadas (Bubble Sort): [9.94, 9.93, 9.93, 9.93, 9.93, 9.92, 9.92, 9.82, 9.8,
.82, 9.8, 9.74]
Top 10 notas ordenadas (sorted()): [9.94, 9.93, 9.93, 9.93, 9.93, 9.92, 9.92, 9.82, 9.8, 9.7
4]
Tiempo Bubble Sort: 0.023373 segundos
Tiempo sorted(): 0.000000 segundos
```

Luego de ejecutar el código, vemos que es mucho más rápido usar la función sorted() que definir una función, y cuanto más grande la lista, mejor se ve esta diferencia.

Metodología Utilizada

El desarrollo del presente trabajo se llevó a cabo siguiendo una metodología de tipo práctico-exploratoria, orientada a comprender y reproducir el comportamiento de procesos sospechosos en un entorno Linux controlado. A continuación, se detallan las etapas y recursos empleados a lo largo del proyecto.

1. Investigación previa

Como primer paso, se realizó una revisión teórica y técnica sobre el tema elegido que fue el ordenamiento de listas o datos. Las fuentes consultadas incluyeron:

- Python Software Foundation. (2024). *Python documentation: Built-in Functions*
- TUPaD-P1-Integ. Búsqueda y ordenamiento
- ChatGPT

2. Diseño y desarrollo del código

Generar 500 notas aleatorias (como si fueran de alumnos).

Aplicar dos formas de ordenamiento de mayor a menor:

- Una con bubble_sort_desc() (ordenamiento burbuja descendente).
- Otra con sorted() usando reverse=True.

Medir y comparar los tiempos de ejecución.

Mostrar los 10 mejores promedios ordenados con cada método.

3. Herramientas y recursos utilizados

Se utilizó Visual Studio Code, para realizar las pruebas, las bibliotecas random (para crear la lista aleatoria) y time (para medir el tiempo) y los comandos explicados anteriormente.

4.Trabajo colaborativo:

Hemos trabajado todo el tiempo en equipo, tanto en las pruebas de código, como en el armado del trabajo, para el cual se creó un archivo de Word compartido para poder ir editando y armando el trabajo de la forma más comprensible y didáctica posible.

Resultados Obtenidos

En este trabajo se implementaron y compararon dos métodos de ordenamiento: el algoritmo de burbuja (Bubble Sort) y la función integrada sorted() de Python.

✅Ambos se aplicaron a una lista de 500 notas generadas aleatoriamente, simulando un caso real de ordenamiento de promedios escolares.

✅Los resultados mostraron que la función sorted() es considerablemente más eficiente que Bubble Sort. Mientras que sorted() ordenó la lista en una fracción de segundo, el algoritmo de burbuja tardó mucho más tiempo debido a su estructura menos optimizada. Además, sorted() ofrece mayor flexibilidad, permitiendo ordenar fácilmente en forma descendente o por criterios personalizados, lo que la hace más adecuada para aplicaciones reales.

✅Esta comparación evidencia la importancia de elegir el algoritmo correcto según el contexto. Aunque Bubble Sort es útil para aprender lógica de programación, no es recomendable para listas grandes o situaciones donde el rendimiento es importante.

Las dificultades encontradas en el desarrollo del código fueron:

❌ En la definición de la funcion bubble sort, no lográbamos el orden descendente, lo que fue solucionado con una comparación entre los valores.

❌ En un principio se ejecutó el código directamente con la lista notas, pero al estar ya modificada por la primera función, no se cumplía el objetivo de la comparación. Para solucionarlo se crearon dos listas iguales, pero con diferente nombre para poder utilizarlas en las diferentes funciones de igual forma y así poder comparar.

Conclusiones

El desarrollo de este trabajo permitió comprender la importancia de los algoritmos de ordenamiento dentro del campo de la programación, tanto desde una perspectiva teórica como práctica. A través de la comparación entre el algoritmo de tipo burbuja y la función sorted() de Python, se observaron diferencias significativas en cuanto a rendimiento, eficiencia y aplicabilidad.

Si bien algoritmos como Bubble Sort son útiles para introducir conceptos de lógica algorítmica, su desempeño es limitado ante volúmenes de datos grandes. Por el contrario, funciones integradas como sorted() demuestran ser herramientas poderosas y versátiles, adaptadas a las necesidades reales del desarrollo de software actual.

Esta experiencia resalta la importancia de elegir el algoritmo adecuado según el contexto del problema, así como de aprovechar las funcionalidades que ofrece un lenguaje como Python. Se cumplió el objetivo de analizar y aplicar distintas formas de ordenamiento, generando un aprendizaje práctico y significativo.

Bibliografía

- Python Software Foundation. (2024). *Python 3 Documentation*. https://docs.python.org/3/
- Python Software Foundation. (2024). *Python documentation: Built-in Functions*
- TUPaD-P1-Integ.- Búsqueda y ordenamiento
- TUPaD-P1-Integ.- Funciones
- ChatGPT

Marco Teórico: Búsquedas

¿Qué son los algoritmos de búsqueda?

Los algoritmos de búsqueda son métodos diseñados para encontrar un elemento dentro de un conjunto de datos. Por ejemplo, pueden utilizarse para encontrar una palabra en una lista o en una base de datos. Estos algoritmos pueden:
- Buscar si un elemento existe.
- Devolver su posición (índice) en la estructura.
- Confirmar su ausencia.

¿Dónde se usan?

Se utilizan en múltiples áreas, como:
- Motores de búsqueda (Google, Bing).
- Videojuegos (IA, rutas).
- Bases de datos (consultas rápidas).
- Aplicaciones de seguridad y sistemas operativos.

Tipos de algoritmos de búsqueda

1. Búsqueda Lineal (o Secuencial)

Es la más simple, pero también la más lenta en listas grandes. Recorre elemento por elemento desde el principio hasta el final.

Funcionamiento:

1. Comienza desde el primer elemento.
2. Compara con el valor buscado.
3. Si no coincide, pasa al siguiente.
4. Se detiene si lo encuentra o termina la lista.

Complejidad:
- Tiempo:
  - Mejor caso: O(1) (si está en la primera posición).
  - Peor caso: O(n) (si está al final o no está).
- Espacio: O(1) (usa muy poca memoria adicional).

2. Búsqueda Binaria

Es más rápida que la búsqueda lineal, pero requiere que los datos estén ordenados (de menor a mayor o viceversa).

Funcionamiento:
1. Toma el elemento del medio.
2. Lo compara con el objetivo.
3. Si el valor buscado es menor, busca en la mitad izquierda.
4. Si es mayor, busca en la mitad derecha.
5. Repite el proceso hasta encontrarlo o quedarse sin elementos.

Complejidad:
- Tiempo:
  - Mejor caso: O(1) (si está en el centro).
  - Peor caso: O(log n).
- Espacio: O(1) (versión iterativa).

3. Búsqueda por Interpolación

También requiere que los datos estén ordenados, pero es más eficiente que la binaria cuando los valores están distribuidos uniformemente (ej.: [10, 20, 30, 40, 50]).

Funcionamiento:
- Utiliza una fórmula matemática (parecida a una regla de tres) para estimar dónde está el valor, en lugar de mirar siempre al medio.
- Si el valor estimado no coincide, ajusta el rango y repite el cálculo.

Complejidad:
- Promedio: O(log log n)
- Peor caso: O(n) (si la distribución no es uniforme)

4. Búsqueda Hash (Tabla Hash)

Usa una estructura llamada tabla hash (o diccionario en Python), que asocia claves con valores. Es la forma más rápida de búsqueda promedio.

Funcionamiento:
1. Se aplica una función hash a la clave.
2. La función devuelve un índice en la tabla.
3. El valor se guarda (o busca) directamente en ese índice.

Ventajas:
- Tiempo promedio de búsqueda: O(1) (muy rápido).

Desventajas:
- Puede haber colisiones: cuando dos claves distintas generan el mismo índice.
- En ese caso, se usan técnicas para resolverlas (como encadenamiento o sondeo lineal).

Descripción del Caso Práctico

El presente trabajo práctico tiene como objetivo aplicar conceptos fundamentales de algoritmos de ordenamiento y búsqueda en programación. Se desarrolló un programa en Python que permite al usuario ingresar 20 números enteros, los cuales son luego ordenados de menor a mayor utilizando el algoritmo Bubble Sort. Una vez ordenada la lista, el usuario puede buscar un número específico mediante búsqueda binaria, un método eficiente que requiere que los datos estén previamente ordenados.

Además, se implementó la medición del tiempo de ejecución para ambas operaciones (ordenamiento y búsqueda), con el fin de analizar el rendimiento de los algoritmos utilizados. Esto permite observar en la práctica las diferencias de eficiencia entre los distintos procesos.

Este caso práctico simula una situación común en programación: tener una colección de datos, ordenarlos para facilitar su manejo y posteriormente buscar información de forma rápida. A través de esta experiencia, se afianzan conocimientos teóricos sobre algoritmos y estructuras de datos, y se desarrollan habilidades prácticas para su implementación.

Resultados Obtenidos

Luego de ejecutar el programa, se pudo comprobar el correcto funcionamiento tanto del algoritmo de ordenamiento (*Bubble Sort*) como del algoritmo de búsqueda (*búsqueda binaria*). El código permitió ingresar exactamente 20 números enteros, los cuales fueron ordenados de menor a mayor sin errores.

A modo de prueba, se ingresaron valores y el programa devolvió correctamente la lista ordenada

Posteriormente, se probó la búsqueda binaria.

Además, se midieron los tiempos de ejecución de ambas operaciones.

Todo se muestra en el video del caso práctico.

Esto confirma que el algoritmo de ordenamiento es considerablemente más costoso en términos de tiempo que el de búsqueda, lo cual era esperable.

En todos los casos de prueba realizados, el programa funcionó correctamente, respetando las condiciones impuestas (20 números exactos, entrada válida, y búsqueda precisa sobre lista ordenada). También se verificó que, al ingresar una cantidad distinta de números, el programa emite un mensaje de advertencia y finaliza correctamente.

Conclusiones del Trabajo Práctico

En este trabajo implementé dos algoritmos fundamentales: Bubble Sort para ordenar y Búsqueda Binaria para buscar un número en la lista. Ambos conceptos son importantes para entender cómo funcionan internamente los programas que manejan grandes volúmenes de datos.

- El algoritmo Bubble Sort, aunque es fácil de entender y programar, no es eficiente para listas grandes. Se puede notar que su tiempo de ejecución aumenta rápidamente cuando la cantidad de datos crece.
- La búsqueda binaria, por otro lado, es muy eficiente, pero solo se puede usar si la lista está ordenada previamente. Esto justifica la necesidad de ordenar antes de buscar.

- Medir el tiempo de ejecución me permitió visualizar la diferencia de rendimiento entre ordenar y buscar**.** Mientras que el ordenamiento tomó más tiempo, la búsqueda fue prácticamente instantánea.
- Este ejercicio me ayudó a comprender mejor cómo se relacionan el ordenamiento y la búsqueda en la programación, y también a poner en práctica buenas prácticas como la validación de entradas y el uso de funciones.
- Como mejora futura, se podrían utilizar algoritmos más eficientes como Quick Sort o Merge Sort para ordenar, y comparar sus tiempos con Bubble Sort.

Bibliografía

Python.org – Documentación oficial
https://docs.python.org/3/

https://www.geeksforgeeks.org/python-program-for-bubble-sort/

https://tup.sied.utn.edu.ar/mod/resource/view.php?id=3434

https://www.youtube.com/watch?v=gJlQTq80lIg

https://www.youtube.com/watch?v=xntUhrhtLaw