

Trabajo Práctico Integrador

“Algoritmos de Búsqueda y Ordenamiento”

Alumnos: Wolanink, Melany Luz Valeria (wolaninkmelany@gmail.com)
Gomez, Ezequiel Horacio (ezequielhgomez811@gmail.com)

Tecnicatura Universitaria En Programación

Universidad Tecnológica Nacional

Programación I

Comisión: 23 / 25

Docente Titular: Nicolas Quirós

Docente Tutor: Flor Camila Gubiotti

Fecha de entrega: 27 de junio de 2025

Índice

Seccion	Pagina
1. Portada	1
2. Índice	2
3. Introducción	3
4. Marco Teórico	4
5. Caso Práctico	8
6. Metodología Utilizada	11
7. Resultados Obtenidos	12
8. Conclusiones	12
9. Bibliografía	13
10. Anexos	14

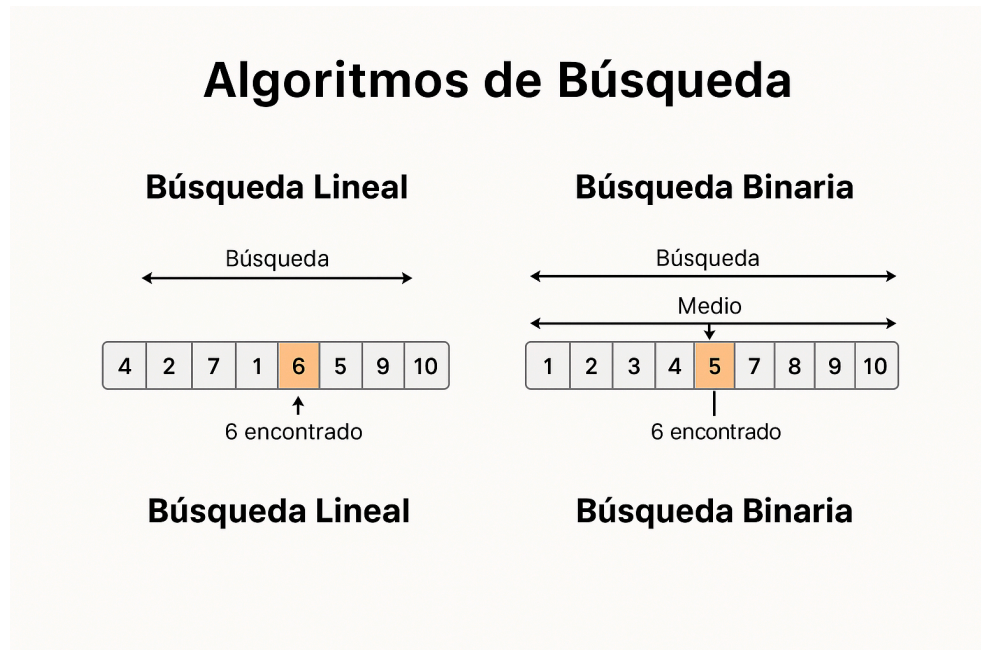
Introducción

En el mundo de la programación, el manejo eficiente de datos es una habilidad fundamental. Cada vez que un programa necesita encontrar un elemento dentro de una colección o reordenar información para mejorar su accesibilidad, entran en juego los algoritmos de búsqueda y ordenamiento. Estos algoritmos forman parte de la base del desarrollo de software, ya que permiten optimizar recursos y mejorar el rendimiento de las aplicaciones. En este trabajo se presentan y analizan distintos métodos clásicos de búsqueda y ordenamiento implementados en Python, con el objetivo de comprender su funcionamiento, eficiencia y casos de uso más adecuados.

¿Por qué se eligió este tema?

Elegimos trabajar con algoritmos de búsqueda y ordenamiento porque son conceptos esenciales en la programación y se aplican constantemente en la resolución de problemas reales. Desde organizar listas hasta encontrar información específica en bases de datos, estos algoritmos permiten desarrollar software más rápido, eficiente y escalable. Además, representan una excelente oportunidad para afianzar conocimientos sobre estructuras de control, lógica algorítmica y análisis de eficiencia, utilizando Python como lenguaje de apoyo. Este tema también nos pareció ideal para desarrollar un caso práctico con resultados fácilmente medibles y visuales.

Marco Teórico



Algoritmos de Búsqueda: Un algoritmo de búsqueda es una serie de pasos que se usan para encontrar un dato específico dentro de una estructura, como una lista, un arreglo o un árbol. Su objetivo principal es localizar un valor determinado, ya sea para saber si está presente o para usarlo dentro de otro proceso. Estos algoritmos se utilizan constantemente en programación, desde buscar un número en una lista hasta encontrar rutas o palabras clave en sistemas más complejos.

Existen distintos tipos de algoritmos de búsqueda, cada uno con sus ventajas y limitaciones según el tipo de datos o el problema a resolver.

- **Búsqueda Lineal**

Es la forma más simple de búsqueda. Consiste en recorrer todos los elementos uno por uno hasta encontrar el valor buscado o llegar al final. No necesita que la lista esté ordenada, por eso es muy útil cuando trabajamos con datos sin estructura.

Sin embargo, es poco eficiente para listas grandes, ya que en el peor caso revisa todos los elementos. Su complejidad es $O(n)$.

- **Búsqueda Binaria**

Es más eficiente que la búsqueda lineal, pero solo funciona si la lista está previamente ordenada. Lo que hace es dividir la lista a la mitad y comparar el elemento central con el que se quiere encontrar. Si no lo encuentra, se queda con la mitad donde puede estar y vuelve a repetir el proceso.

Este algoritmo es mucho más rápido en listas grandes, ya que su complejidad es $O(\log n)$.

- **Búsqueda en Grafos o Árboles**

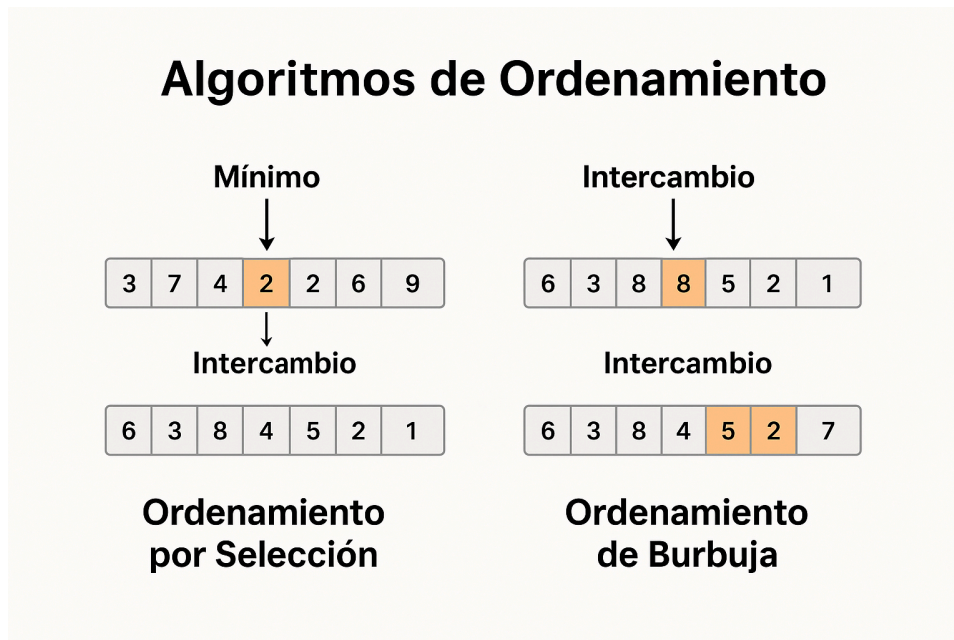
Cuando los datos están organizados en forma de árbol o grafo, se pueden usar algoritmos como Búsqueda en Anchura (BFS) o Búsqueda en Profundidad (DFS).

BSF recorre por niveles (de izquierda a derecha), mientras que DFS baja por una rama lo más posible antes de volver atrás. Estos métodos son muy comunes en inteligencia artificial, laberintos, mapas o sistemas jerárquicos.

- **Búsqueda con Tablas Hash**

Otro tipo de búsqueda es la que se hace usando estructuras como [tablas hash](#), donde cada elemento se ubica en una posición calculada a partir de su valor. Esto permite encontrar datos casi instantáneamente, con una eficiencia promedio de $O(1)$

El único inconveniente es que puede haber colisiones (dos valores que van al mismo lugar), y eso requiere técnicas adicionales para resolverlo.



Algoritmos de Ordenamiento: Los algoritmos de ordenamiento se utilizan para re-organizar datos dentro de una estructura (como una lista o un arreglo) de forma que queden en un orden específico, generalmente ascendente o descendente. Ordenar datos es una tarea común en programación, ya que facilita búsquedas más rápidas, comparaciones, visualizaciones y análisis más eficientes.

Existen muchos algoritmos para ordenar, cada uno con ventajas y desventajas según el tamaño de los datos, si ya están parcialmente ordenados o si el orden debe ser estable (que mantenga posiciones relativas de los elementos iguales).

Ordenamiento Burbuja (Bubble Sort)

Es uno de los algoritmos más conocidos y simples. Funciona comparando pares de elementos adyacentes e intercambiándose si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista está completamente ordenada.

- Muy fácil de entender e implementar.
- Poco eficiente en listas grandes.

Ordenamiento por Selección (Selection Sort)

Este algoritmo recorre la lista para encontrar el valor más pequeño y lo coloca en la primera posición. Luego repite el proceso con el resto de la lista. No cambia de lugar los elementos si no es necesario.

También tiene mal rendimiento en listas grandes.

Ordenamiento por Inserción (Insertion Sort)

Funciona de forma similar a cómo uno ordenaría a mano cartas de una baraja. Toma un elemento de la lista y lo inserta en su lugar correcto dentro de una sub lista ya ordenada.

- Muy eficiente para listas pequeñas o casi ordenadas.
- Complejidad: $O(n^2)$, pero $O(n)$ en el mejor de los casos (lista casi ordenada).

Quicksort

Es un algoritmo muy eficiente en la práctica. Elige un **pivote**, separa los elementos en dos sublistas (menores y mayores al pivote) y luego aplica recursivamente el mismo proceso en cada sublista. Finalmente, une los resultados.

- Muy usado en software real.
- Rápido, pero sufre si la lista ya está muy ordenada sin pivote aleatorio.
- Complejidad promedio: $O(n \log n)$; peor caso: $O(n^2)$

Mergesort

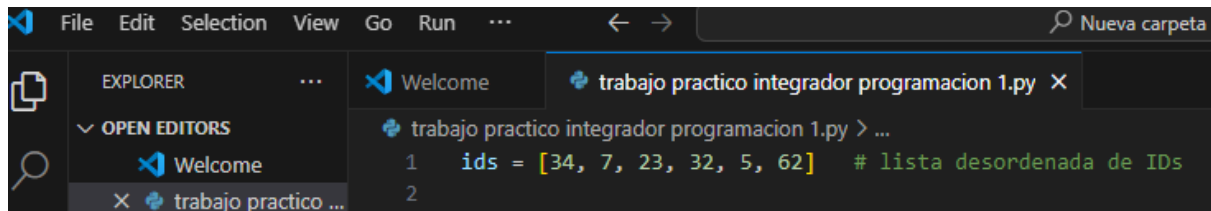
Divide la lista en mitades, ordena cada mitad y luego las fusiona (merge) de forma ordenada. Es muy predecible y eficiente, incluso en el peor caso.

- Usa más memoria que otros, pero es muy confiable

Caso Práctico

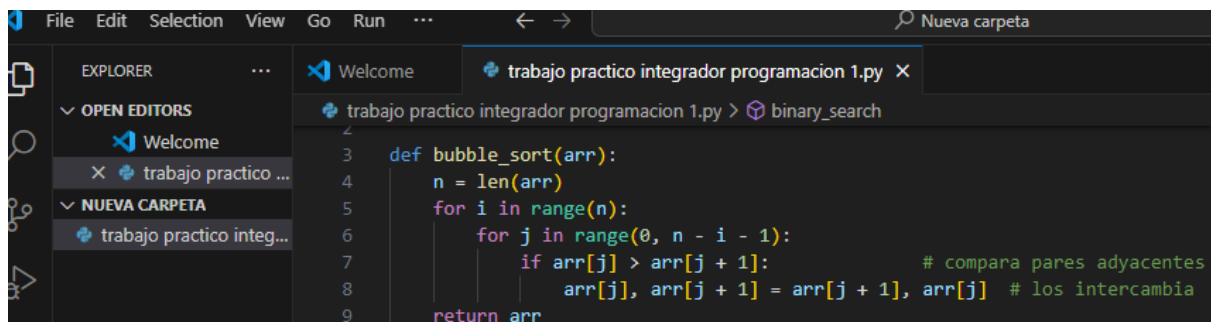
Imaginamos que cada número representa el **ID de un libro** en el sistema.

1. Datos de partida:



```
File Edit Selection View Go Run ... Nueva carpeta
EXPLORER
OPEN EDITORS
Welcome
trabajo practico integrador programacion 1.py X
trabajo practico integrador programacion 1.py > ...
1 ids = [34, 7, 23, 32, 5, 62] # lista desordenada de IDs
2
```

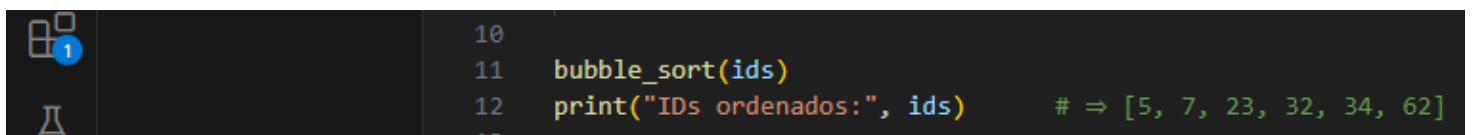
2. Bubble Sort (ordenar):



```
File Edit Selection View Go Run ... Nueva carpeta
EXPLORER
OPEN EDITORS
Welcome
trabajo practico integrador programacion 1.py X
trabajo practico integrador programacion 1.py > binary_search
2
3 def bubble_sort(arr):
4     n = len(arr)
5     for i in range(n):
6         for j in range(0, n - i - 1):
7             if arr[j] > arr[j + 1]: # compara pares adyacentes
8                 arr[j], arr[j + 1] = arr[j + 1], arr[j] # los intercambia
9     return arr
```

- a) Recorre la lista **n veces**.
- b) Compara e intercambia elementos vecinos si están desordenados.
- c) Resultado: la lista queda ordenada “in-place”.

3) Ordenar la lista:



```
10
11 bubble_sort(ids)
12 print("IDs ordenados:", ids) # => [5, 7, 23, 32, 34, 62]
13
```


4) Búsqueda binaria:

```
13
14 def binary_search(arr, target):
15     low, high = 0, len(arr) - 1
16     while low <= high:
17         mid = (low + high) // 2
18         if arr[mid] == target:           # encontrado
19             return mid
20         if arr[mid] < target:           # buscar en la mitad derecha
21             low = mid + 1
22         else:                           # buscar en la mitad izquierda
23             high = mid - 1
24     return -1                           # no encontrado
```

a) Define los extremos `low` y `high`.

b) Cálculo del medio `mid`.

c) Compara:

Si es igual : devuelves índice;

Si es menor: descarta mitad izquierda;

Si es mayor: descarta la mitad derecha.

Repite hasta encontrar o agotar la búsqueda.

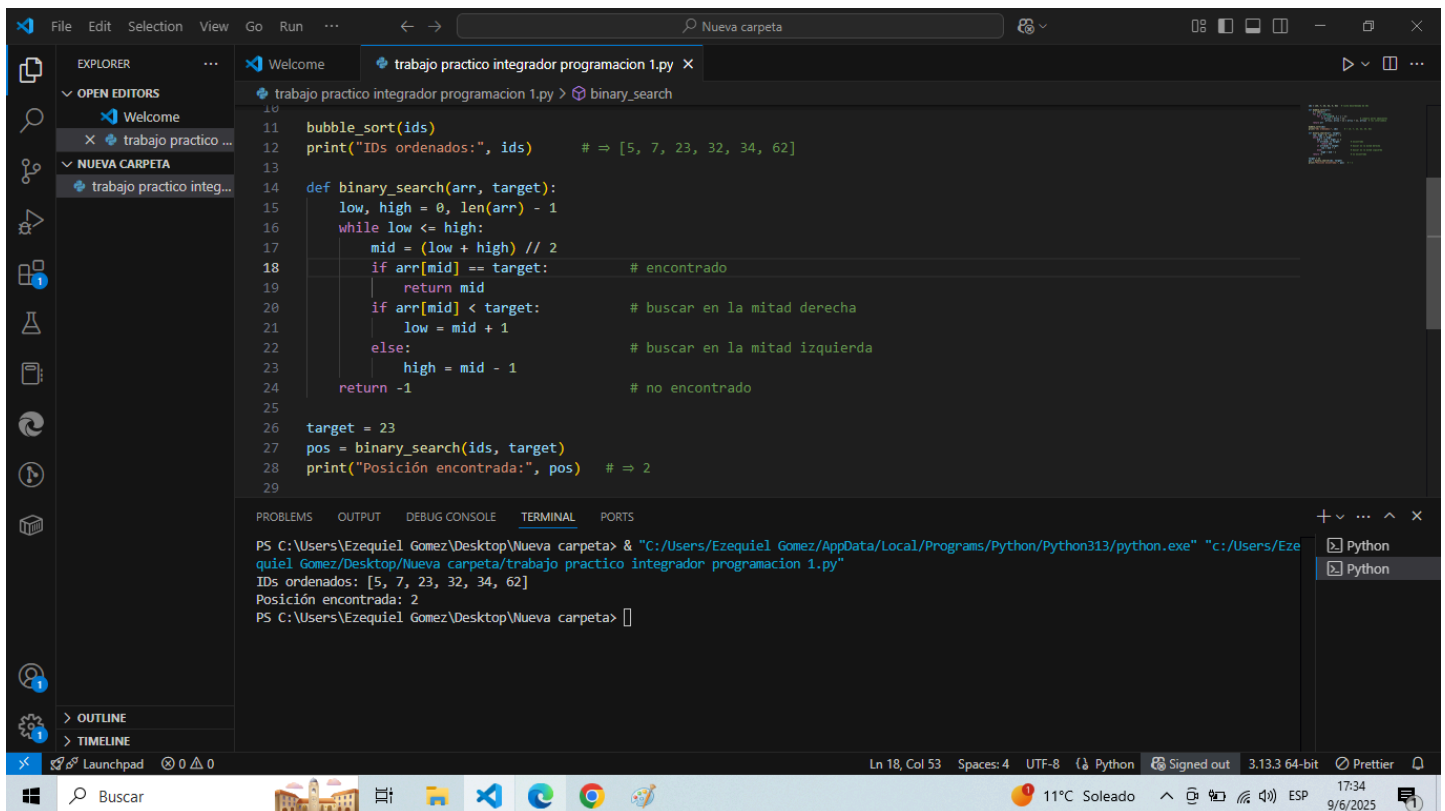
5) Buscar un ID concreto:

```

26 target = 23
27 pos = binary_search(ids, target)
28 print("Posición encontrada:", pos) # => 2

```

Si devuelve **-1** significa que el ID no existe en la lista.



The screenshot shows a Visual Studio Code editor with a Python file named 'trabajo practico integrador programacion 1.py'. The code implements a bubble sort and a binary search function. The terminal output shows the execution of the script, which sorts the list [5, 7, 23, 32, 34, 62] and finds the position of the target value 23 at index 2.

```

10
11 bubble_sort(ids)
12 print("IDs ordenados:", ids) # => [5, 7, 23, 32, 34, 62]
13
14 def binary_search(arr, target):
15     low, high = 0, len(arr) - 1
16     while low <= high:
17         mid = (low + high) // 2
18         if arr[mid] == target: # encontrado
19             return mid
20         if arr[mid] < target: # buscar en la mitad derecha
21             low = mid + 1
22         else: # buscar en la mitad izquierda
23             high = mid - 1
24     return -1 # no encontrado
25
26 target = 23
27 pos = binary_search(ids, target)
28 print("Posición encontrada:", pos) # => 2
29

```

Terminal Output:

```

PS C:\Users\Ezequiel Gomez\Desktop\Nueva carpeta> "C:/Users/Ezequiel Gomez/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/Ezequiel Gomez/Desktop/Nueva carpeta/trabajo practico integrador programacion 1.py"
IDs ordenados: [5, 7, 23, 32, 34, 62]
Posición encontrada: 2
PS C:\Users\Ezequiel Gomez\Desktop\Nueva carpeta>

```

Metodología Utilizada

La elaboración del presente trabajo se desarrolló en varias etapas bien definidas:

- Recolección de información teórica: se investigaron conceptos fundamentales sobre algoritmos de búsqueda y ordenamiento a través de fuentes confiables como documentación oficial, libros y artículos especializados
- Implementación en Python: se codificaron los algoritmos seleccionados utilizando el lenguaje Python, aplicando buenas prácticas de programación y estructura modular.
- Pruebas con diferentes conjuntos de datos: se realizaron pruebas utilizando listas pequeñas, medianas y grandes, tanto ordenadas como desordenadas, para analizar el comportamiento y eficiencia de cada algoritmo.
- Registro de resultados y validación: se documentaron los resultados obtenidos durante las pruebas, evaluando la precisión, tiempos de ejecución y condiciones de funcionamiento.
- Elaboración del informe y anexos: se redactó el presente informe, incluyendo el marco teórico, el caso práctico, conclusiones y gráficos explicativos, acompañados del código fuente y capturas del funcionamiento.

Resultados Obtenidos

En este trabajo se desarrolló una solución básica en Python para ordenar una lista de IDs de libros y luego buscar un ID específico dentro de ella. Se utilizaron dos algoritmos clásicos: Bubble Sort para el ordenamiento y Búsqueda Binaria para localizar valores dentro de la lista ya ordenada.

Conclusión

Realizar este trabajo nos ayudó a entender en profundidad cómo funcionan los algoritmos de búsqueda y ordenamiento, no solo desde la teoría, sino también aplicándolos en código real. Pudimos ver las diferencias entre cada uno, cuándo conviene usar uno u otro y cómo afecta el tipo de datos a la eficiencia del programa.

Además, al implementarlos en Python y probar con distintos conjuntos de datos, reforzamos nuestro conocimiento sobre estructuras como listas, funciones, y tiempos de ejecución. También aprendimos a organizar mejor nuestro código, documentarlo y validar que realmente funcione como esperábamos.

En general, fue una experiencia muy útil para afianzar lo que venimos aprendiendo en la materia y para darnos cuenta de la importancia que tienen estos algoritmos en la programación de todos los días.

Beneficios:

Mejora la organización de datos.

- Aumenta la velocidad de búsqueda.
- Código simple y reutilizable.
- Base sólida para futuros algoritmos.
- Refuerza buenas prácticas de programación.

Bibliografía

1. Geeks for Geeks.
 1. <https://www.geeksforgeeks.org/sorting-algorithms/>
 2. <https://www.geeksforgeeks.org/searching-algorithms/>
2. Programiz.

<https://www.programiz.com/dsa/sorting-algorithm>
3. TutorialsPoint.
 1. https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm
 2. https://www.tutorialspoint.com/search_algorithm
4. Wikipedia contributors. (2025). Wikipedia.
https://en.wikipedia.org/wiki/Sorting_algorithm
5. Python Software Foundation. (2024).
<https://docs.python.org/3/library/bisect.html>
6. Real Python. (n.d.).

<https://realpython.com/binary-search-python/>

Anexos

- Repositorio en GitHub:
<https://github.com/ezequielgomez2025/TPIPROGRAMACION1>
- Video explicativo: https://youtu.be/bsD_QYpIPEE