UNIVERSIDAD TECNOLÓGICA NACIONAL

TRABAJO PRÁCTICO

INTEGRADOR

PROGRAMACIÓN I

Tema: Algoritmos de búsqueda y ordenamiento

Alumnos:

- Gastón Armando Giorgio - gastongiorgio26@gmail.com

- Alfredo Alberto de Inocenti - alfreddinocenti@gmail.com

Materia: Programación I

Profesor/a: Julieta Trapé

Fecha de entrega: 9 de junio de 2025

ÍNDICE

1. INTRODUCCIÓN

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales en la programación y el manejo eficiente de datos. Su estudio permite comprender cómo seleccionar la técnica adecuada de acuerdo con el contexto y el volumen de información a procesar.

¿POR QUÉ SE ELIGIÓ ESTE TEMA?

- **Importancia en la programación:** los algoritmos de búsqueda y ordenamiento son pilares en la ciencia de la computación y la programación, siendo la base de la eficiencia en el manejo de datos. Son conceptos que todo programador debe dominar.
- **Presentes en la tecnología moderna:** estos algoritmos están presentes en casi todas las aplicaciones y sistemas que usamos a diario. Por ejemplo, al buscar un contacto en el teléfono, ordenar resultados en una tienda online, o incluso en la forma en que los datos se organizan en una base de datos.
- **Optimización y rendimiento:** la elección del algoritmo correcto puede marcar una diferencia abismal en el rendimiento de un programa, especialmente al trabajar con grandes volúmenes de datos.
- **Desafío intelectual:** el análisis de la complejidad de los algoritmos y la búsqueda de soluciones óptimas nos resulta un desafío intelectual atractivo y fundamental para nuestra formación como programador.

¿QUÉ IMPORTANCIA TIENE EN LA PROGRAMACIÓN?

- **Eficiencia y escalabilidad:** son cruciales para escribir código eficiente y escalable. Un algoritmo mal elegido puede llevar a programas lentos o que colapsan con grandes cantidades de datos. Por ejemplo, ordenar una lista de millones de elementos con un algoritmo $O(n^2)$ sería inviable, mientras que uno O(n log n) lo haría en segundos.

- **Base para estructuras de datos:** Los algoritmos de búsqueda y ordenamiento están íntimamente ligados a las estructuras de datos. Muchas estructuras (árboles binarios de búsqueda, tablas hash) se basan en principios de búsqueda y ordenamiento para su funcionamiento óptimo.

- **Aplicaciones prácticas diversas:** existen varios ejemplos concretos en donde son aplicados:

  - **Bases de datos:** consultas eficientes, indexación de datos.

- **Inteligencia Artificial/Machine Learning:** procesamiento de datos, algoritmos de clustering.
- **Motores de búsqueda:** organizar y clasificar resultados de búsqueda.
- **Gráficos por computadora:** renderizado y organización de objetos.
- **Bioinformática:** alineación de secuencias de ADN.
- **Juegos:** lógica de enemigos, pathfinding.

❖ **Desarrollo de habilidades de pensamiento lógico:** El estudio de estos algoritmos fomenta el pensamiento crítico y la capacidad de resolver problemas de manera estructurada y eficiente.

¿QUÉ OBJETIVOS SE PROPONE ALCANZAR CON EL DESARROLLO DE ESTE TRABAJO?

Los objetivos principales que se tuvieron en cuenta son:

- Comprender los **principios fundamentales** de los algoritmos de búsqueda (ej. lineal, binaria) y ordenamiento (ej. burbuja, selección, inserción, quicksort).
- Analizar la **complejidad temporal y espacial** de los algoritmos seleccionados.
- Implementar al menos dos algoritmos de **búsqueda** y cuatro de **ordenamiento** en lenguaje **Python**.
- Comparar el rendimiento de los algoritmos implementados a través de un **caso práctico**.
- Identificar las **situaciones óptimas** para el uso de cada algoritmo.

2. MARCO TEÓRICO

En el mundo del desarrollo de software, los algoritmos de búsqueda y ordenamiento juegan un papel fundamental, estas técnicas permiten organizar y obtener datos de una manera muy eficiente, lo que es esencial para optimizar el rendimiento de las aplicaciones.[1]

¿Qué son los algoritmos de búsqueda?

Los algoritmos de búsqueda son métodos que nos permiten encontrar la ubicación de un elemento específico dentro de una lista de elementos. Dependiendo de la lista necesitarás utilizar un algoritmo u otro, por ejemplo, si la lista tiene elementos ordenados, puedes usar un algoritmo de búsqueda

[1] Página 4geek (2025). Título: *"Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos"*. Extraído de: https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python.

binaria, pero si la lista contiene los elementos de forma desordenada este algoritmo no te servirá, para buscar un elemento en una lista desordenada deberás utilizar un algoritmo de búsqueda lineal.

La búsqueda es importante en programación porque se utiliza en una amplia variedad de aplicaciones. Algunos ejemplos de uso de la búsqueda en programación son: [2]

- **Búsqueda de palabras clave en un documento:** Se puede utilizar un algoritmo de búsqueda para encontrar todas las apariciones de una palabra clave en un documento.
- **Búsqueda de archivos en un sistema de archivos:** Se puede utilizar un algoritmo de búsqueda para encontrar un archivo con un nombre específico en un sistema de archivos.
- **Búsqueda de registros en una base de datos:** Se puede utilizar un algoritmo de búsqueda para encontrar un registro con un valor específico en una base de datos.
- **Búsqueda de la ruta más corta en un gráfico:** Se puede utilizar un algoritmo de búsqueda para encontrar la ruta más corta entre dos nodos en un gráfico.
- **Búsqueda de soluciones a problemas de optimización:** Se puede utilizar un algoritmo de búsqueda para encontrar soluciones a problemas de optimización, como encontrar el valor máximo de una función.

ALGORITMOS DE BÚSQUEDA

➢ BÚSQUEDA LINEAL[3]

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implican recorrer una lista de elementos uno por uno hasta encontrar un elemento específico. Este algoritmo es muy sencillo de implementar en código, pero puede ser muy ineficiente dependiendo del largo de la lista y la ubicación donde está el elemento.

Se recorre desde el inicio una lista, comparando cada uno de los elementos con el elemento a buscar, en caso de encontrarse se devuelve el índice donde encontró el elemento a buscar o verdadero, eso depende del caso de uso.[4]

[2] Material de estudio de la materia Programación I (2025). Universidad Tecnológica Nacional. Título: "Búsqueda y Ordenamiento en Programación".
[3] Página 4geek (2025). Título: *"Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos"*. Extraído de: https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python.
[4] Blog Medium (2020). Andrés Mise Olivera. Título: *"Algoritmos de Búsqueda y Ordenamiento"*. Extraído de: https://medium.com/@mise/algoritmos-de-b%C3%BAsqueda-y-ordenamiento-7116bcea03d0.

Es poco usada debido a que existen algoritmos más óptimos para realizar búsquedas, es útil cuando se desea encontrar un elemento en un conjunto de datos pequeño y la ganancia de ordenarlo es poca vs realizar la búsqueda directamente.

Complejidad temporal de la búsqueda lineal[5]

- **Peor caso: O(n)**, donde n es el número de elementos en la lista. Esto ocurre cuando el elemento buscado está al final de la lista o no está presente.
- **Mejor caso: O(1)**, cuando el elemento buscado es el primero de la lista.
- **Caso promedio: O(n)**, porque en promedio se recorren la mitad de los elementos.

Ventajas y desventajas del algoritmo de búsqueda lineal[6]

❖ **Ventajas:**

- **Sencillez**: La búsqueda lineal es uno de los algoritmos de búsqueda más simples y fáciles de implementar. Solo requiere iterar a través de la lista de elementos uno por uno hasta encontrar el objetivo.
- **Flexibilidad**: La búsqueda lineal puede aplicarse a cualquier tipo de lista, independientemente de si está ordenada o no.

❖ **Desventajas:**

- **Ineficiencia en listas grandes**: La principal desventaja de la búsqueda lineal es su ineficiencia en listas grandes. Debido a que compara cada elemento uno por uno, su tiempo de ejecución crece de manera lineal con el tamaño de la lista.
- **No es adecuada para listas ordenadas**: Aunque puede funcionar en listas no ordenadas, la búsqueda lineal no es eficiente para listas ordenadas. En tales casos, algoritmos de búsqueda más eficientes, como la búsqueda binaria, son preferibles.

[5] Archivo jupyter notebook (2025). Universidad Tecnológica Nacional. Título: *"Clase de Programación: Búsquedas y Ordenamiento en Python".*
[6] Página 4geek (2025). Título: *"Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos"*. Extraído de: https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python.

- **BÚSQUEDA BINARIA**

El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias.[7]

Teniendo una lista de datos ordenada se divide en dos, compara el valor a buscar con el que se encuentra en la mitad, si el valor coincide termina la búsqueda, en caso que no coincida comienza un proceso repetitivo donde el rango a buscar depende si el valor a buscar es mayor o menor al que se encuentra en la mitad. Si el valor a buscar es menor al de la mitad, el rango de búsqueda va a ser desde el inicio hasta el de la mitad; de lo contrario, si el número a buscar es mayor a la mitad, el rango de búsqueda es desde la mitad hasta el último dato de la lista, esto se repite hasta que se encuentre el valor o el intervalo de búsqueda este vacío.[8]

Complejidad temporal de la búsqueda binaria[9]

- **Peor caso: O(log n)**, donde n es el número de elementos en la lista. Esto ocurre cuando el elemento no está presente o está en una de las divisiones finales.
- **Mejor caso: O(1)**, cuando el elemento buscado está justo en el centro de la lista.
- **Caso promedio: O(log n)**, porque el algoritmo divide la lista en mitades en cada iteración.

Ventajas y desventajas del algoritmo de búsqueda binaria

- **Ventajas:**

  - **Eficiencia de listas ordenadas**: La principal ventaja de la búsqueda binaria es su eficiencia en listas ordenadas. Su tiempo de ejecución es de **O(log n)**, lo que significa que disminuye rápidamente a medida que el tamaño de la lista aumenta.

  - **Menos comparaciones**: Comparado con la búsqueda lineal, la búsqueda binaria realiza menos comparaciones en promedio, lo que lo hace más rápido para encontrar el objetivo.

[7] Página 4geek (2025). Título: *"Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos"*. Extraído de: https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python.
[8] Blog Medium (2020). Andrés Mise Olivera. Título: *"Algoritmos de Búsqueda y Ordenamiento"*. Extraído de: https://medium.com/@mise/algoritmos-de-b%C3%BAsqueda-y-ordenamiento-7116bcea03d0.
[9] Archivo jupyter notebook (2025). Universidad Tecnológica Nacional. Título: *"Clase de Programación: Búsquedas y Ordenamiento en Python".*

❖ **Desventajas:**

- **Requiere una lista ordenada**: La búsqueda binaria sólo es aplicable a listas ordenadas, Si la lista no está ordenada, se debe realizar una operación adicional para ordenarla antes de usar la búsqueda binaria.
- **Mayor complejidad de implementación**: Comparado con la búsqueda lineal, la búsqueda binaria es más compleja de implementar debido a su naturaleza recursiva.

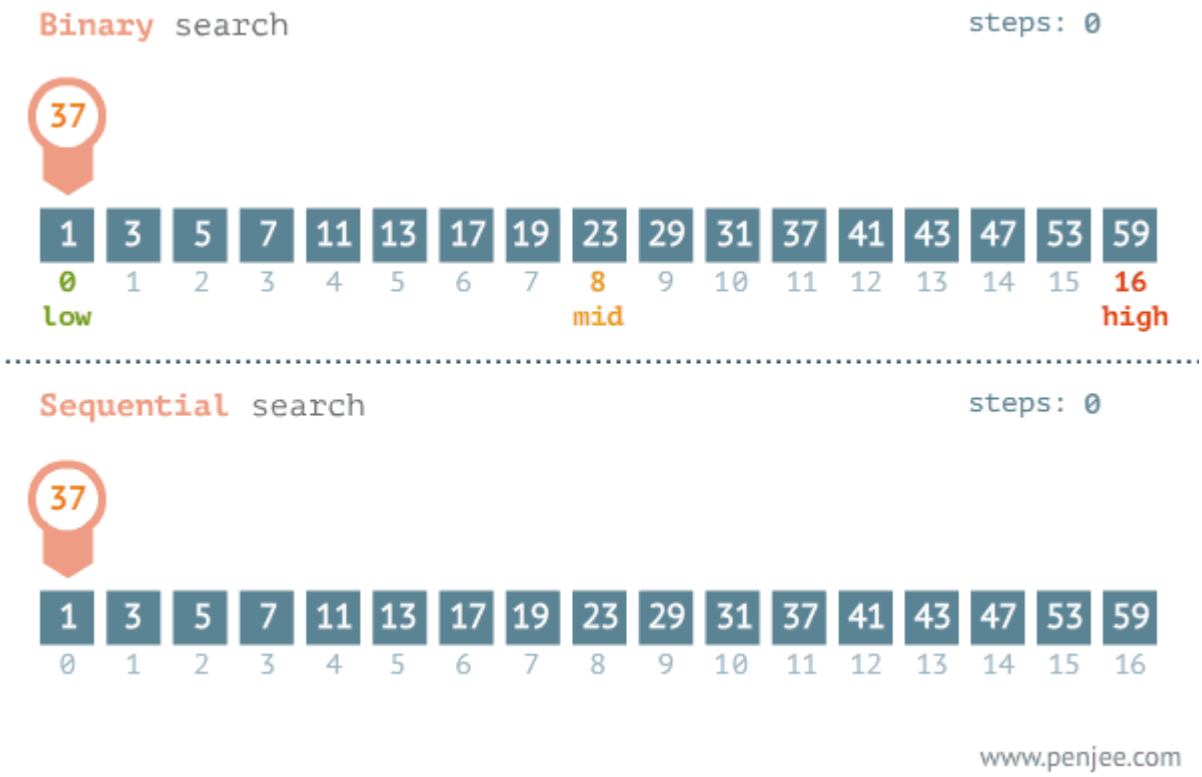


Figura 1. Ejemplos visuales de búsqueda lineal y binaria. Recuperado de: https://medium.com/@mise/algoritmos-de-b%C3%BAsqueda-y-ordenamiento-7116bcea03d0.

Cuadro comparativo entra búsqueda lineal y binaria

Algoritmo	Descripción	Complejidad temporal	Requisito previo
Búsqueda lineal	Recorre secuencialmente la colección hasta encontrar el elemento o agotar la lista.	O(n) peor caso/caso promedio O(1) mejor caso	Ninguno
Búsqueda binaria	Divide iterativamente la lista ordenada en mitades descartando la que no puede contener el objetivo.	O(log n) peor caso O(1) mejor caso	Lista ordenada

La eficiencia de la búsqueda binaria radica en eliminar la mitad de las posibilidades en cada iteración O(log n).

¿Qué son los algoritmos de ordenamiento?[10]

En la informática, los algoritmos de ordenamiento son cruciales para la optimización de una tarea, estos permiten organizar datos de manera que puedan ser accedidos y utilizados de manera más eficiente. Un algoritmo de ordenamiento permite reorganizar una lista de elementos o nodos en un orden específico, por ejemplo, de forma ascendente o descendente dependiendo de la ocasión.

ALGORITMOS DE ORDENAMIENTO

➢ ORDENAMIENTO DE BURBUJA (BUBBLE SORT)[11]

El ordenamiento de burbuja es el algoritmo de ordenamiento más simple. Funciona intercambiando repetidamente los elementos adyacentes si están en el orden incorrecto. Este algoritmo no es adecuado para grandes conjuntos de datos, ya que su complejidad temporal promedio y en el peor de los casos es bastante alta.

[10] Página 4geek (2025). Título: *"Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos"*. Extraído de: https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python.
[11] Página geeksforgeeks (2024). Título: "*Ordenamiento por selección*". Extraído y traducido al español de: https://www.geeksforgeeks.org/selection-sort-algorithm-2/

1. **Ordenamos la lista mediante múltiples pasadas.** Tras la primera, el elemento más grande va al final (su posición correcta). De igual manera, tras la segunda, el segundo elemento más grande va a la penúltima posición, y así sucesivamente.
2. **En cada pasada, procesamos solo los elementos que aún no se han movido a la posición correcta.** Después de k pasadas, los k elementos más grandes deben haberse movido a las últimas k posiciones.
3. En una pasada, consideramos los elementos restantes y comparamos todos los adyacentes, **intercambiando si el elemento mayor está antes que el menor.** Si continuamos haciendo esto, obtenemos el elemento mayor (entre los restantes) en su posición correcta.

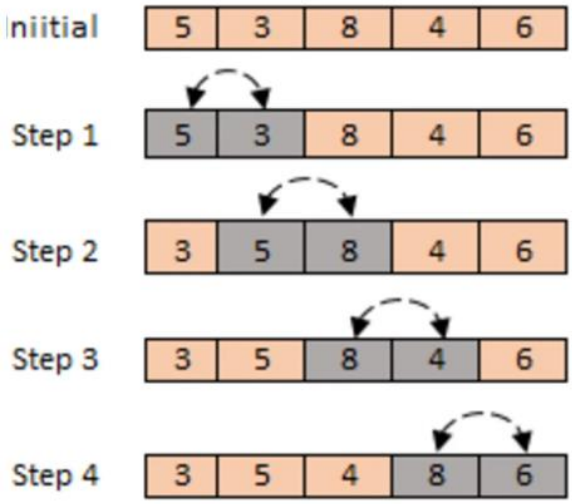


Figura 2. Ejemplo visual de ordenamiento por burbuja (2023). Recuperado de: https://programacionpython80889555.wordpress.com/2023/01/26/algoritmos-de-ordenacion-en-python-ordenamiento-de-burbuja/

Complejidad temporal[12]

- **Peor caso: O(n^2)**, donde n es el número de elementos en la lista. Esto ocurre cuando la lista está en orden inverso.
- **Mejor caso: O(n)**, cuando la lista ya está ordenada (con una optimización que detecta si no hubo intercambios).
- **Caso promedio: O(n^2)**.

[12] Archivo jupyter notebook (2025). Universidad Tecnológica Nacional. Título: *"Clase de Programación: Búsquedas y Ordenamiento en Python".*

Ventajas y desventajas del ordenamiento de burbuja (Bubble Sort):

- **Ventajas:**
  - **Simplicidad**: El algoritmo de burbuja es fácil de entender e implementar, lo que lo convierte en una buena opción para introducir conceptos de ordenamiento en la programación.
  - **Implementación sencilla**: Requiere poca cantidad de código y no involucra estructuras de datos complejas.
- **Desventajas:**
  - **Lento para listas grandes**: Debido a su complejidad cuadrática $O(n^2)$, el algoritmo de burbuja se vuelve lento en la práctica para listas de tamaño considerable.
  - **No considera el orden parcial**: A diferencia de otros algoritmos, el algoritmo de burbuja realiza el mismo número de comparaciones e intercambios sin importar si la lista ya está en gran parte ordenada.

➢ ORDENAMIENTO POR INSERCIÓN (INSERTION SORT)[13]

El ordenamiento por inserción es un algoritmo de ordenación simple que funciona insertando iterativamente cada elemento de una lista desordenada en su posición correcta dentro de la sección ordenada de la lista. Es como ordenar naipes. Se dividen las cartas en dos grupos: las ordenadas y las desordenadas. Luego, se elige una carta del grupo desordenado y se coloca en el lugar correcto dentro del grupo ordenado.

1. **Comenzamos con el segundo elemento de la lista**, ya que se supone que el primer elemento está ordenado.
2. **Comparamos el segundo elemento con el primer elemento**; si el segundo elemento es más pequeño, intercámbielos.
3. **Pasamos al tercer elemento**, lo comparamos con los dos primeros elementos y los colocamos en su posición correcta.
4. **Repetimos** hasta que toda la matriz esté ordenada.

[13] Página geeksforgeeks (2024). Título: "*Algoritmo de ordenación por inserción*". Extraído y traducido al español de: https://www.geeksforgeeks.org/insertion-sort-algorithm/

Figura 3. Ejemplo visual de ordenamiento por inserción (2020). Recuperado de: https://binarycoffee.dev/post/insertion-sort

Complejidad temporal:

- **Peor caso**: **O(n²)**, donde *n* es el número de elementos en la lista. Ocurre cuando la lista está en orden inverso, porque cada nuevo elemento debe compararse y moverse a la primera posición.
- **Mejor caso**: **O(n)**, cuando la lista ya está ordenada. En este caso, solo se realizan *n-1* comparaciones y ningún movimiento.
- **Caso promedio**: **O(n²)**. En una lista con elementos en orden aleatorio, cada elemento se compara con aproximadamente la mitad de los elementos anteriores.

Ventajas y Desventajas del Ordenamiento por inserción (Insertion Sort)

❖ **Ventajas:**

- **Baja sobrecarga**: requiere menos comparaciones y movimientos que algoritmos como el ordenamiento de burbuja, lo que lo hace más eficiente en términos de intercambios de elementos.
- **Simplicidad**: el ordenamiento por inserción es uno de los algoritmos de ordenamiento más simples de implementar y entender. Esto lo hace adecuado para enseñar conceptos básicos de ordenamiento.

❖ **Desventajas:**

- **Ineficiencia en listas grandes**: a medida que el tamaño de la lista aumenta, el rendimiento del ordenamiento por inserción disminuye. Su complejidad cuadrática de **O(n^2)** en el peor caso lo hace ineficiente para las listas grandes.
- **No escalable**: al igual que otros algoritmos de complejidad cuadrática, el ordenamiento por inserción no es escalable para listas grandes, ya que su tiempo de ejecución aumenta considerablemente con el tamaño de la lista.

➢ ORDENAMIENTO POR SELECCIÓN (SELECTION SORT)[14]

El ordenamiento por selección es un algoritmo de ordenamiento basado en la comparación. Ordena una lista seleccionando repetidamente el elemento más pequeño de la parte no ordenada e intercambiándolo con el primer elemento no ordenado. Este proceso continúa hasta que toda la lista esté ordenada.

1. **Primero encontramos el elemento más pequeño y lo intercambiamos con el primero.** De esta manera, el elemento más pequeño queda en su posición correcta.
2. **Luego encontramos el más pequeño entre los elementos restantes** (o el segundo más pequeño) y lo intercambiamos con el segundo elemento.
3. **Seguimos haciendo esto** hasta que consigamos mover todos los elementos a la posición correcta.

[14] Página geeksforgeeks (2024). Título: "Ordenamiento por selección". Extraído y traducido al español de: https://www.geeksforgeeks.org/selection-sort-algorithm-2/

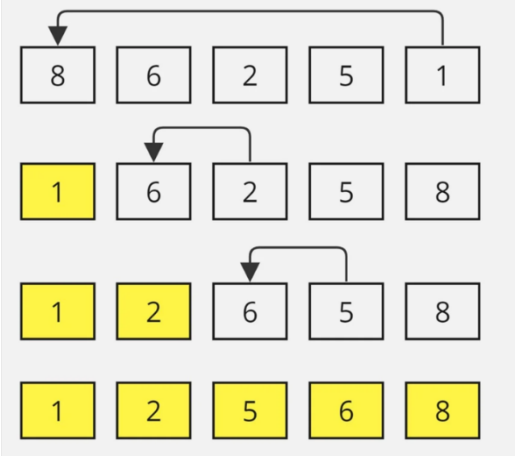


Figura 4. Ejemplo visual de ordenamiento por selección (2025). Recuperado de: https://www.venkys.io/articles/details/selection-sort

Complejidad temporal:

- **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.
- **Caso promedio: $O(n^2)$**

Ventajas y desventajas del algoritmo de selección:

❖ **Ventajas:**

- Fácil de entender e implementar, lo que lo hace ideal para enseñar conceptos básicos de clasificación.
- Requiere únicamente un espacio de memoria extra constante O(1).
- Requiere menos intercambios (o escrituras en memoria) en comparación con muchos otros algoritmos estándar.

❖ **Desventajas:**

- La ordenación por selección tiene una complejidad temporal de **$O(n^2)$**, lo que la hace más lenta en comparación con algoritmos como Quick Sort o Merge Sort.
- No mantiene el orden relativo de elementos iguales lo que significa que no es estable.

➢ ORDENAMIENTO QUICKSORT[15]

Quicksort es un algoritmo eficiente que utiliza la técnica de "divide y vencerás". Funciona seleccionando un elemento pivote y particionando la lista en dos sublistas: una con elementos menores que el pivote y otra con elementos mayores. Luego, se aplica Quicksort recursivamente a cada sublista.

Cuando usar: generalmente es más eficiente que inserción y selección para listas grandes, especialmente cuando se implementa de manera eficiente.

El algoritmo consta principalmente de cuatro pasos:[16]

1. **Elegir un pivote:** seleccionar un elemento de la lista como pivote. La elección del pivote puede variar (ej.: primer elemento, último elemento, elemento aleatorio o mediana).
2. **Particionar la lista:** reorganizar la lista alrededor del pivote. Tras la partición, todos los elementos menores que el pivote estarán a su izquierda y todos los mayores que el pivote estarán a su derecha. El pivote estará entonces en su posición correcta y obtenemos su índice.
3. **Llamada recursiva:** aplicar recursivamente el mismo proceso a las dos sublistas particionadas (izquierda y derecha del pivote).
4. **Caso base:** la recursión se detiene cuando solo queda un elemento en la sublista, ya que ya hay un solo elemento ordenado.

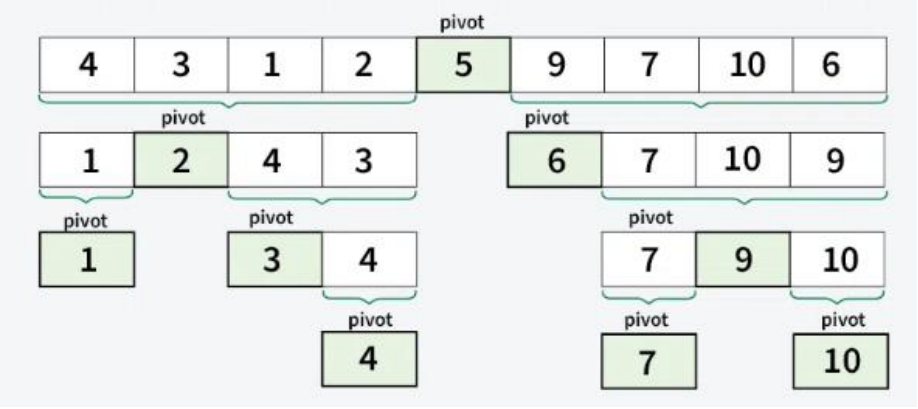


Figura 5. Ejemplo visual de ordenamiento quick sort. Extraído de: https://ronnyml.wordpress.com/2009/07/19/quicksort-en-c/

[15] Archivo jupyter notebook (2025). Universidad Tecnológica Nacional. Título: *"Clase de Programación: Búsquedas y Ordenamiento en Python".*

[16] Página geeksforgeeks (2024). Título: "*Quick Sort*". Extraído y traducido al español de: https://www.geeksforgeeks.org/quick-sort-algorithm/

Complejidad temporal:

- **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)**.

Cuadro comparativo de algoritmos de ordenamiento

Algoritmo	Enfoque	Mejor caso	Promedio	Peor caso	Complejidad espacial
Bubble Sort	Comparaciones e intercambios adyacentes	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	Inserta el elemento en una lista ya ordenada	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	Selecciona el mínimo elemento, y lo coloca en orden	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Quick Sort	Divide por pivote, ordena recursivamente	O(n log n)	O(n log n)	$O(n^2)$	O(n log n)

La Quick Sort se considera generalmente más rápida en la práctica para colecciones grandes desordenadas, mientras que Insertion Sort suele ser la opción preferida para colecciones pequeñas o casi ordenadas.

3. CASO PRÁCTICO

Para la demostración y análisis de los algoritmos de ordenamiento, se desarrolló un caso práctico que simula un escenario común en el comercio electrónico: **la necesidad de ordenar un catálogo de productos por precio.**

El sistema, implementado en un Jupyter Notebook, genera una lista de productos ficticios para una tienda online.

Cada producto es representado como un diccionario de Python con los siguientes campos:

- **id**: Un identificador numérico único para cada producto.
- **nombre**: El nombre del producto (ej: "Televisor", "Celular", "Laptop").
- **precio**: Un valor numérico con dos decimales, generado de forma aleatoria.

El objetivo principal de este caso práctico es aplicar los diferentes algoritmos de ordenamiento a esta lista de productos y medir su rendimiento para determinar cuál es el más eficiente a medida que aumenta la cantidad de productos en el catálogo.

4. METODOLOGÍA UTILIZADA

Para llevar a cabo la comparación de rendimiento, se siguió una metodología estructurada en los siguientes pasos:

1. **Importación de librerías (ver Anexo A.1):** se importaron todas las utilidades necesarias para que el resto del programa pueda funcionar y realizar sus tareas, especialmente aquellas que implican simulación, análisis de tiempo y visualización de datos.
   - `import random`: trae la capacidad de generar números y selecciones aleatorias.
   - `import time`: permite trabajar con el tiempo, lo que podría usarse para medir el rendimiento de algoritmos o para introducir pausas.
   - `import matplotlib.pyplot as plt`: importa una herramienta poderosa para crear gráficos y visualizar datos, ideal para mostrar tendencias o resultados de simulaciones.
   - `import copy`: Incluye funcionalidades para duplicar objetos complejos, como listas de diccionarios, asegurando que las modificaciones hechas a una copia no afecten la versión original, lo cual es vital para realizar pruebas comparativas sin alterar los datos fuente.

2. **Generación de Datos (ver Anexo B.1)**: Se creó una función en Python llamada **`generar_productos`** que produce listas de diccionarios (productos) de diferentes tamaños.

   Esta función utiliza una metodología de **generación de datos simulados** para crear una lista de productos. Funciona iterando la cantidad de veces solicitada, y en cada paso, construye un **diccionario** que representa un producto individual. Los atributos de este producto, como el **nombre** y el **precio**, se asignan de forma **aleatoria** a partir de conjuntos predefinidos o rangos, mientras que el id es consecutivo.

   Cada diccionario de producto se agrega a una **lista**, y finalmente, la función devuelve esta lista completa, simulando un conjunto de datos de productos para diversos usos.

3. **Implementación de Algoritmos (ver Anexos C.1 a C.4)**:

   Se implementaron en Python cuatro **algoritmos de ordenamiento** para organizar la lista de productos en función de su clave precio en orden ascendente:

   - **Bubble Sort (Burbuja)**
   - **Selection Sort (Selección)**
   - **Insertion Sort (Inserción)**
   - **QuickSort (Rápido)**

   Además, se implementaron los **algoritmos de búsqueda lineal y binaria** en Python, para encontrar un producto según su número de ID.

   Para preparar los datos, se generaron listas de productos de diferentes tamaños (n) para simular un escenario de tienda online. Es crucial destacar que, para la búsqueda binaria, las listas generadas deben estar previamente ordenadas para garantizar la correcta aplicación del algoritmo.

   Para cada tamaño de lista n, se realizaron varias búsquedas de elementos que sí existen en posiciones aleatorias dentro de la lista. Se utilizó un cronómetro para medir el tiempo que tardaba cada algoritmo en encontrar el elemento.

   Para obtener resultados más representativos y disminuir la variabilidad en las mediciones individuales, se promediaron los tiempos resultantes de las múltiples búsquedas realizadas para cada tamaño de lista.

Los tiempos promedio obtenidos se graficaron en función del tamaño de la lista (n) para visualizar el comportamiento de cada algoritmo a medida que el volumen de datos aumenta.

4. **Medición de Tiempos de Ejecución (ver Anexos D.1 a D.2)**: Se basa en dos funciones clave:
   - **`medir_tiempo_ordenamiento`**: Esta función toma un algoritmo de ordenamiento, una lista de datos original y un número de repeticiones. Su metodología es repetir la ejecución del algoritmo múltiples veces sobre una copia profunda (para asegurar que cada prueba empiece con una lista desordenada idéntica) de la lista original.

     Mide el tiempo de inicio y fin con `time.time()` y almacena las diferencias.

     Finalmente, retorna el **tiempo promedio de todas las ejecuciones**, lo que ayuda a mitigar las variaciones transitorias del sistema.

   - **`generar_y_medir_ordenamiento`**: Esta función orquesta el proceso de evaluación. Primero, itera a través de diferentes tamaños de lista predefinidos. Para cada tamaño, genera una nueva lista de productos.

     Luego, para cada algoritmo de ordenamiento proporcionado, invoca a `medir_tiempo_ordenamiento` para obtener su tiempo promedio.

     Todos estos resultados se almacenan en un diccionario estructurado por algoritmo y tamaño de lista, permitiendo una comparación clara de su rendimiento a medida que los datos aumentan.

5. **Visualización de Resultados (ver Anexo E.1 a F.1)**: Los tiempos obtenidos para cada algoritmo en los distintos tamaños de lista se recopilaron y se utilizaron para generar varios gráficos de líneas mediante la biblioteca matplotlib.pyplot.
   Estos gráficos permiten una comparación visual directa del rendimiento y la escalabilidad de cada algoritmo.

5. RESULTADOS OBTENIDOS

El análisis comparativo del rendimiento de los algoritmos de ordenamiento, visualizado en el gráfico generado, arrojó resultados claros y consistentes con la teoría de la complejidad algorítmica **(ver Anexo F.2)**.

- **QuickSort**: Demostró ser, por un amplio margen, el algoritmo **más eficiente** de los cuatro. Su tiempo de ejecución creció de manera mucho más lenta (logarítmica) en comparación con los otros, mostrando una excelente escalabilidad. Incluso con 5000 productos, su tiempo de ejecución fue significativamente bajo.

- **Bubble Sort, Selection Sort e Insertion Sort**: Estos tres algoritmos mostraron un comportamiento similar entre sí, con un rendimiento considerablemente inferior al de QuickSort. Sus tiempos de ejecución aumentaron de forma cuadrática a medida que crecía el tamaño de la lista. Esto significa que, al duplicar la cantidad de datos, el tiempo necesario para ordenarlos se cuadruplicaba aproximadamente, lo que los hace ineficientes para grandes volúmenes de información, como se evidencia en la pronunciada curva ascendente del gráfico.

En resumen, los resultados prácticos confirmaron que para una tarea como ordenar un catálogo de productos, donde la cantidad de elementos puede ser grande, la elección de un algoritmo eficiente como **QuickSort** es crucial para el rendimiento del sistema.

Los resultados de la simulación de los **algoritmos de búsqueda lineal y binaria** se presentan a continuación **(Ver Anexos G.1 a H.4)**:

- **Tiempos de ejecución:** Con los tamaños de lista elegidos para la simulación y el hardware utilizado, los tiempos individuales de búsqueda fueron del orden de 10^{-8} segundos. Es decir, una cienmillonésima parte de un segundo (1 segundo dividido por 100.000.000).

- **Solapamiento de líneas en el gráfico:** Debido a la muy baja magnitud de los tiempos de ejecución y la resolución del cronómetro, las líneas que representan el rendimiento de la búsqueda lineal y binaria en el gráfico se mostraron prácticamente superpuestas cuando se realizó en el mejor caso (el elemento estaba en la primera posición). Esto indica que, para los volúmenes de datos probados en esta simulación y la precisión de la medición, la diferencia de rendimiento entre ambos algoritmos no fue perceptible visualmente en el gráfico.

- **Efectividad del algoritmo de búsqueda binaria:** sin embargo, cuando el volumen de datos de entrada aumento significativamente, es decir, cuando se pudo probar en el peor de los casos (cuando el elemento no estaba en la lista), se pudo apreciar visualmente la **gran efectividad del algoritmo de búsqueda binaria** por sobre la búsqueda lineal.

6. CONCLUSIONES

Este trabajo práctico ha permitido validar los conceptos teóricos de los algoritmos de ordenamiento y búsqueda de forma práctica. La implementación y posterior evaluación de su rendimiento en un caso práctico demostraron la importancia crítica de la eficiencia y la escalabilidad en la programación.

- Algoritmos de ordenamiento:

La diferencia de rendimiento entre los algoritmos de complejidad **$O(n^2)$** (Bubble, Selection, Insertion) y el de complejidad **O(n log n)** (QuickSort) fue contundente. Mientras que los primeros son sencillos de implementar, su uso en aplicaciones reales con conjuntos de datos medianos o grandes es inviable y puede llevar a sistemas lentos y poco responsivos.

QuickSort, por su parte, se destaca como una solución robusta y eficiente para este tipo de problemas, justificando por qué es uno de los algoritmos de ordenamiento más utilizados en la industria del software.

- Algoritmos de búsqueda:

A pesar del solapamiento de las líneas en el gráfico de tiempos, se pueden extraer las siguientes conclusiones fundamentales sobre los algoritmos de búsqueda lineal y binaria:

Eficiencia teórica vs. práctica en pequeña escala: en teoría, la búsqueda lineal tiene una complejidad de **O(n)**, lo que significa que su tiempo de ejecución crece linealmente con el tamaño de la lista.

Por otro lado, la búsqueda binaria tiene una complejidad de **O(log n)**, lo que indica un crecimiento logarítmico mucho más lento.

Sin embargo, en la práctica, para listas de tamaños reducidos y con hardware moderno, la eficiencia es tan alta que la diferencia entre ambos algoritmos es mínima y no se evidencia claramente en mediciones de tan corta duración.

Importancia del volumen de datos: si bien en esta simulación los tiempos son casi idénticos, es crucial entender que, en condiciones reales, con listas de productos significativamente más grandes (por ejemplo, miles o millones de elementos) o en escenarios donde se acumulen una gran cantidad de repeticiones de búsqueda, la diferencia de rendimiento se volvería dramáticamente evidente.

Comportamiento esperado en listas grandes:

- La **búsqueda lineal** tenderá a mostrar un crecimiento con una pendiente cercana a 1, es decir, su tiempo de ejecución aumentará proporcionalmente al número de elementos en la lista.

- La **búsqueda binaria**, por su parte, apenas mostrará un aumento en su tiempo de ejecución a medida que el tamaño de la lista crece, debido a su naturaleza logarítmica. Esto la hace exponencialmente más eficiente para grandes volúmenes de datos.

Aunque en alguna de las simulaciones las líneas se solapan, **la búsqueda binaria demuestra su ventaja a partir de unos pocos miles de elementos o cuando se requiere una gran cantidad de búsquedas repetidas, debido a su eficiencia superior en términos de complejidad algorítmica.** La elección entre un algoritmo y otro dependerá del contexto específico de la aplicación, el volumen de datos a manejar y los requisitos de rendimiento.

Para listas pequeñas, la simplicidad de la búsqueda lineal puede ser suficiente, pero para sistemas con grandes bases de datos, la búsqueda binaria (siempre que los datos estén ordenados) es indispensable para garantizar un rendimiento óptimo.

El desarrollo de este trabajo cumplió con los objetivos propuestos, ya que no solo se implementaron los algoritmos, sino que se analizó su rendimiento y se comprendieron las implicaciones prácticas de su complejidad, reforzando la idea de que un buen programador no solo debe saber cómo resolver un problema, sino cómo hacerlo de la manera más eficiente posible.

7. BIBLIOGRAFÍA

A continuación, se presenta la bibliografía citada en el marco teórico del trabajo:

- Material de estudio de la materia Programación I (2025). Universidad Tecnológica Nacional. Título: “Búsqueda y Ordenamiento en Programación”.
- Página 4geek (2025). Título: “Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos”.
- Blog Medium (2020). Andrés Mise Olivera. Título: “Algoritmos de Búsqueda y Ordenamiento”.
- Archivo jupyter notebook (2025). Universidad Tecnológica Nacional. Título: “Clase de Programación: Búsquedas y Ordenamiento en Python”.
- Página geeksforgeeks (2024). Título: “Ordenamiento por selección”.
- Página geeksforgeeks (2024). Título: “Quick Sort”.

8. ANEXOS

Anexo A.1 - Importación de librerías necesarias:

```python
import random
import time
import matplotlib.pyplot as plt
import copy # Usaremos copy.deepcopy para no modificar la lista original en cada prueba
```

Anexo B.1 - Generación de datos (diccionario de productos de ejemplo):

```python
# --- Función para generar una lista de productos (diccionario) ---
def generar_productos(cantidad):
    productos_nombre = ["Televisor", "Celular", "Laptop", "Auricular", "Tablet", "Monitor", "Reloj", "Consola", "Cámara", "Altavoz"]
    lista_de_productos = [] # Creamos una lista vacía para guardar nuestros productos
    for i in range(cantidad):
        producto = { # Creamos un diccionario para cada producto
            "id": i + 1, # Un ID único para cada producto
            "nombre": random.choice(productos_nombre), # Un nombre generado aleatoriamente
            "precio": round(random.uniform(50.00, 2000.00), 2) # Un precio aleatorio con dos decimales
        }
        lista_de_productos.append(producto) # Agregamos el diccionario de producto a la lista
    return lista_de_productos

# --- Ejemplo de cómo se vería nuestra lista de productos ---
# Generamos 5 productos para ver su estructura
productos_ejemplo = generar_productos(5)
print("--- Primeros 5 productos de ejemplo generados: ---")
for p in productos_ejemplo:
    print(p)
print("-" * 50)
```

✓ 0.0s

```
--- Primeros 5 productos de ejemplo generados: ---
{'id': 1, 'nombre': 'Auricular', 'precio': 286.23}
{'id': 2, 'nombre': 'Laptop', 'precio': 1847.56}
{'id': 3, 'nombre': 'Consola', 'precio': 492.06}
{'id': 4, 'nombre': 'Tablet', 'precio': 950.0}
{'id': 5, 'nombre': 'Altavoz', 'precio': 266.88}
--------------------------------------------------
```

Anexo C.1 - Ordenamiento de burbuja:

```python
def bubble_sort(lista_productos):
    n = len(lista_productos)

    for i in range(n - 1):
        for j in range(0, n - i - 1):

            if lista_productos[j]['precio'] > lista_productos[j + 1]['precio']:

                lista_productos[j], lista_productos[j+1] = lista_productos[j+1], lista_productos[j]

    return lista_productos
```

Anexo C.2 - Ordenamiento de selección:

```python
def selection_sort(lista_productos):
    n = len(lista_productos)

    for i in range(n):
        min_idx = i

        for j in range(i + 1, n):
            if lista_productos[j]['precio'] < lista_productos[min_idx]['precio']:
                min_idx = j

        lista_productos[i], lista_productos[min_idx] = lista_productos[min_idx], lista_productos[i]

    return lista_productos
```

Anexo C.3 - Ordenamiento de inserción:

```python
def insertion_sort(lista_productos):

    for i in range(1, len(lista_productos)):

        key_producto = lista_productos[i]
        j = i - 1

        while j >= 0 and key_producto['precio'] < lista_productos[j]['precio']:
            lista_productos[j + 1] = lista_productos[j]
            j -= 1

        lista_productos[j + 1] = key_producto

    return lista_productos
```

Anexo C.4 - Ordenamiento quicksort:

```python
def particion(arr, lo, hi):
    pivote = arr[hi]['precio']
    i = lo - 1

    for j in range(lo, hi):
        if arr[j]['precio'] <= pivote:
            i += 1

            tmp = arr[i]
            arr[i] = arr[j]
            arr[j] = tmp

    tmp = arr[i + 1]
    arr[i + 1] = arr[hi]
    arr[hi] = tmp

    return i + 1


def quick_sort(arr, lo: int = 0, hi: int | None = None):
    if hi is None:
        hi = len(arr) - 1

    if lo < hi:
        p = particion(arr, lo, hi)
        quick_sort(arr, lo, p - 1)
        quick_sort(arr, p + 1, hi)

    return arr
```

Anexo D.1 - Función medir tiempo de ordenamiento:

```python
# --- Función para medir el tiempo de ejecución de algoritmos de ordenamiento ---
# Toma una función de algoritmo, una lista original y el número de veces a repetir la prueba.
def medir_tiempo_ordenamiento(algoritmo_func, lista_original, num_repeticiones=20):
    tiempos = [] # Guardaremos los tiempos de cada repetición aquí
    for _ in range(num_repeticiones):
        # Muy importante: Creamos una COPIA profunda de la lista original.
        # Esto asegura que cada algoritmo siempre trabaje con una lista sin ordenar
        # y no afecte las pruebas de los otros algoritmos.
        lista_para_ordenar = copy.deepcopy(lista_original)

        start_time = time.time() # Registramos el tiempo de inicio
        algoritmo_func(lista_para_ordenar) # Ejecutamos el algoritmo de ordenamiento
        end_time = time.time() # Registramos el tiempo de finalización

        tiempos.append(end_time - start_time) # Calculamos el tiempo transcurrido y lo guardamos

    return sum(tiempos) / num_repeticiones # Retornamos el tiempo promedio de todas las repeticiones
```

Anexo D.2 - Función generar y medir ordenamiento:

```python
# --- Función para generar y medir el rendimiento de ordenamiento ---
# Prepara la ejecución de los algoritmos de ordenamiento para diferentes tamaños de lista.
def generar_y_medir_ordenamiento(diccionario_algoritmos, tamanios_lista, num_repeticiones=20):
    # Un diccionario para guardar los tiempos de cada algoritmo para cada tamaño de lista
    resultados_tiempos = {}
    for nombre in diccionario_algoritmos:
        resultados_tiempos[nombre] = []

    for tam in tamanios_lista: # Recorremos los diferentes tamaños de lista
        print(f"-> Generando y midiendo para {tam} productos (Ordenamiento)...")
        # Genera una nueva lista de productos para este tamaño actual
        lista_productos_base = generar_productos(tam)

        for nombre_alg, func_alg in diccionario_algoritmos.items(): # Recorremos cada algoritmo
            # Medimos el tiempo promedio para este algoritmo y este tamaño de lista
            tiempo_promedio = medir_tiempo_ordenamiento(func_alg, lista_productos_base, num_repeticiones)
            resultados_tiempos[nombre_alg].append(tiempo_promedio) # Guardamos el resultado

    return resultados_tiempos # Retornamos todos los resultados de tiempo
```

Anexo E.1 - Función para graficar resultados:

```python
import matplotlib.pyplot as plt
import numpy as np   # asegúrate de haberlo importado antes

# ──────────────────────────────────────────────────────────────
#  Funciones utilitarias de trazado
# ──────────────────────────────────────────────────────────────
def _complejidad_teorica(nombre_algo: str) -> str:
    """Devuelve una etiqueta corta con la complejidad de tiempo esperada."""
    if any(pat in nombre_algo for pat in ('Bubble', 'Insertion', 'Selection')):
        return " (O(n²))"
    if 'Quick' in nombre_algo:
        return " (O(n log n))"
    if 'Lineal' in nombre_algo:
        return " (O(n))"
    if 'Binaria' in nombre_algo:
        return " (O(log n))"
    return ""
```

```python
def graficar_ordenamientos(
        resultados: dict,
        tamanios: list[int],
        titulo: str,
        escala_loglog: bool = False
        ):
    """
    Dibuja curvas de tiempo de algoritmos de ordenamiento.

    resultados : {nombre -> lista de tiempos}
    tamanios   : lista de n usados en los experimentos
    """
    plt.figure(figsize=(8, 5))
    for nombre, tiempos in resultados.items():
        plt.plot(
            tamanios,
            tiempos,
            marker='o',
            markersize=4,
            linewidth=1.4,
            label=nombre + _complejidad_teorica(nombre)
        )
    if escala_loglog:
        plt.xscale('log'); plt.yscale('log')
    plt.title(titulo)
    plt.xlabel("Tamaño de la lista (n)")
    plt.ylabel("Tiempo promedio (s)")
    plt.grid(True, which='both', ls='--', lw=0.5)
    plt.legend(); plt.tight_layout(); plt.show()

```

```python
def graficar_busquedas(
        resultados,
        tamanios,
        titulo,
        escala_loglog=False,
        marcar_break_even=False,
        mostrar_pendientes=False,
        eps=1e-8
        ):                  # ← NUEVO parámetro
    """
    Dibuja curvas de búsqueda.  Si escala_loglog=True,
    sustituye los valores 0 por 'eps' para que no desaparezcan.
    """
    import numpy as np
    plt.figure(figsize=(8, 5))

    # ── Asegurarse de no tener ceros ──────────────────────────
    resultados_plot = {}
    for k, v in resultados.items():
        v_np = np.asarray(v, dtype=float)
        v_np = np.where(v_np <= 0, eps, v_np)      # ← reemplaza 0 por eps
        resultados_plot[k] = v_np

```

```
    # ── Asegurarse de no tener ceros ──────────────────────────
    resultados_plot = {}
    for k, v in resultados.items():
        v_np = np.asarray(v, dtype=float)
        v_np = np.where(v_np <= 0, eps, v_np)      # ← reemplaza 0 por eps
        resultados_plot[k] = v_np

    # ── Curvas de los algoritmos ──────────────────────────────
    for nombre, tiempos in resultados_plot.items():
        plt.plot(
            tamanios,
            tiempos,
            marker='o',
            markersize=4,
            linewidth=1.4,
            label=nombre + _complejidad_teorica(nombre)
        )
```

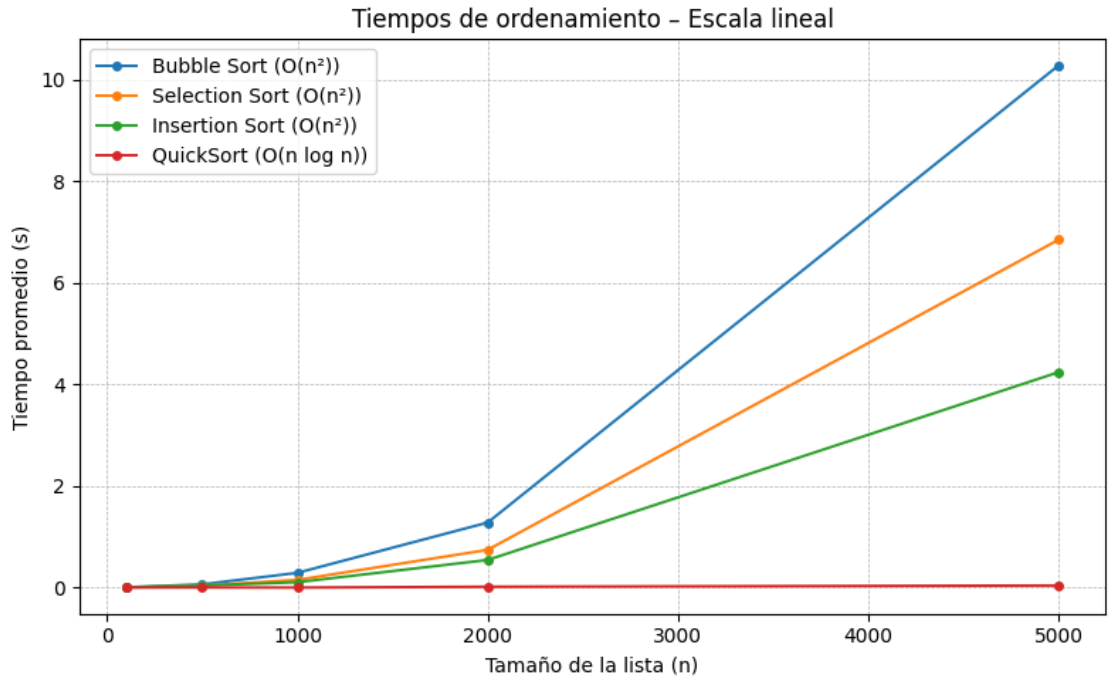
```
    # ── Referencias de pendiente ──────────────────────────────
    if escala_loglog and mostrar_pendientes:
        ref_n = np.array(tamanios)

        # ancla intermedio para que queden en el centro del gráfico
        y_base = np.sqrt(
            resultados_plot[next(iter(resultados_plot))][0] *resultados_plot[next(iter(resultados_plot))][-1]
            )

        plt.plot(
            ref_n,y_base * (ref_n / ref_n[0]),
            ls=':',
            lw=1,
            label='Pendiente 1'
            )

        plt.plot(
            ref_n,y_base * (np.log2(ref_n) / np.log2(ref_n[0])),
            ls='--',
            lw=1,
            label='Pendiente log n'
            )
```

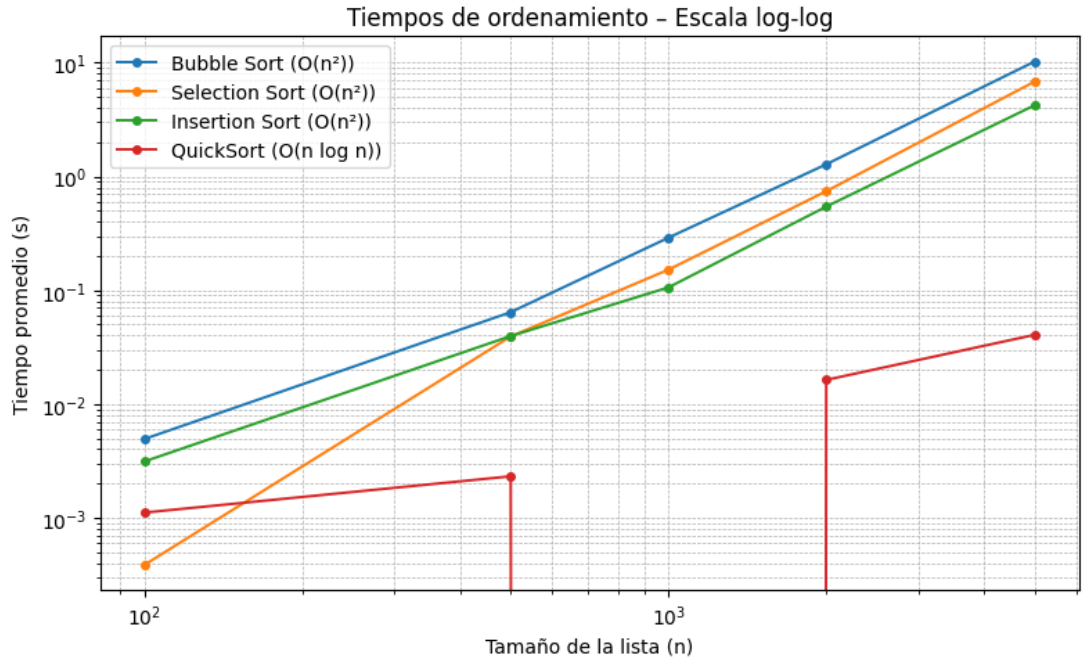
```
    # ── Línea de corte opcional ──────────────────────────────
    if marcar_break_even:
        lineal  = resultados_plot[next(iter(resultados_plot))]
        binaria = resultados_plot[list(resultados_plot.keys())[1]]
        for n, t_lin, t_bin in zip(tamanios, lineal, binaria):
            if t_bin < t_lin:
                plt.axvline(n, color='k', ls='--')
                plt.text(
                    n*1.05,
                    max(lineal)*0.8,
                    f"Binaria ≈ Lineal\nn ≈ {n}",
                    rotation=90,
                    va='top'
                    )
                break

    # ── Escalas y estética ──────────────────────────────────
    if escala_loglog:
        plt.xscale('log'); plt.yscale('log')
    plt.title(titulo)
    plt.xlabel("Tamaño de la lista (n)")
    plt.ylabel("Tiempo promedio (s)")
    plt.grid(True, which='both', ls='--', lw=0.5)
    plt.legend(); plt.tight_layout(); plt.show()

```

Anexo F.1 - Pruebas de ordenamiento:

```
print("--- Iniciando Pruebas de Ordenamiento ---")

# Definir los tamaños de lista a probar.
# Para empezar, podemos usar tamaños más pequeños y luego aumentarlos si QuickSort es muy rápido.
tamanios_para_pruebas_ordenamiento = [100, 500, 1000, 2000, 5000]

# Diccionario que mapea los nombres de los algoritmos a sus funciones
algoritmos_ordenamiento = {
    "Bubble Sort": bubble_sort,
    "Selection Sort": selection_sort,
    "Insertion Sort": insertion_sort,
    "QuickSort": quick_sort
}

# Ejecutar las mediciones para los algoritmos de ordenamiento
resultados_ordenamiento = generar_y_medir_ordenamiento(
    algoritmos_ordenamiento,
    tamanios_para_pruebas_ordenamiento,
    num_repeticiones=5
    )

print("\n--- Graficando Resultados de Ordenamiento ---")

# ──────────────────────────────────────────────────────────
#  Gráficos de ORDENAMIENTO
# ──────────────────────────────────────────────────────────

graficar_ordenamientos(
    resultados_ordenamiento,               # dict {nombre → tiempos}
    tamanios_para_pruebas_ordenamiento,    # lista de n
    "Tiempos de ordenamiento – Escala lineal"
)
```

Anexo F.2 - Gráfico tiempos de algoritmos de ordenamiento:





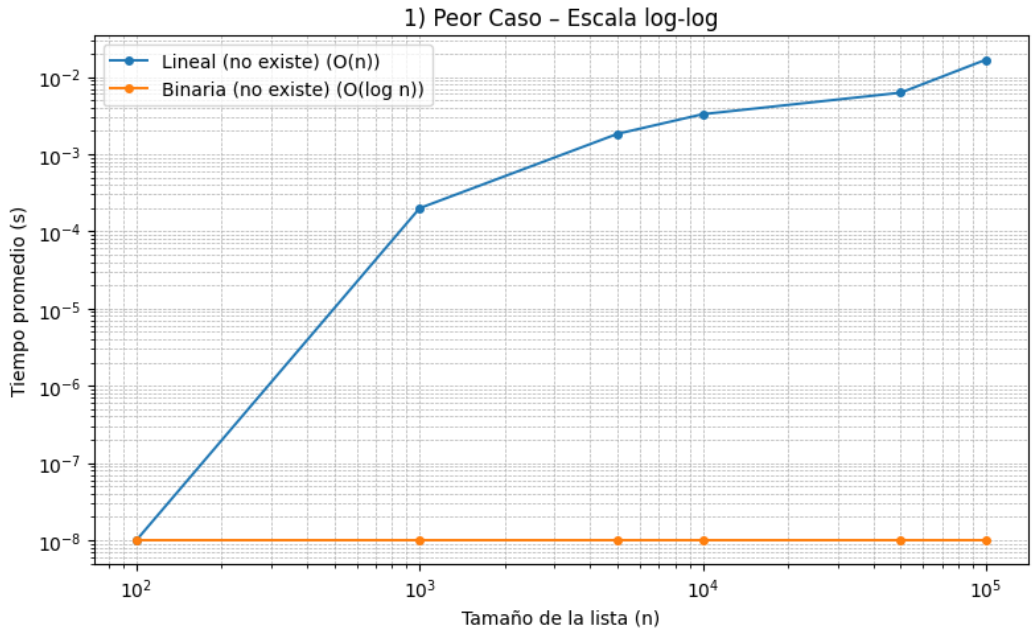
Anexo G.1 - Algoritmo de búsqueda lineal:

```python
def busqueda_lineal_por_nombre(lista_productos, nombre_buscado):
    for producto in lista_productos:
        if producto['nombre'] == nombre_buscado:
            return producto
    return None

def busqueda_lineal_por_id(lista_productos, id_buscado):
    for producto in lista_productos:

        if producto['id'] == id_buscado:
            return producto
    return None

```

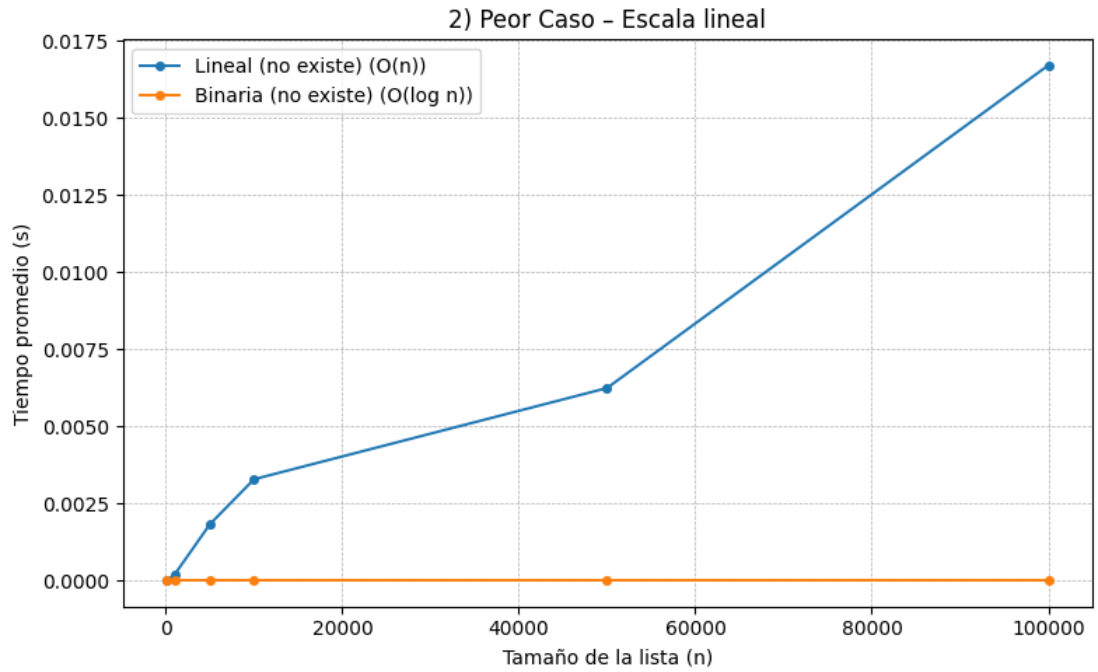
Anexo G.2 - Algoritmo de búsqueda binaria:

```python
def busqueda_binaria_por_id(lista_productos_ordenada_por_id, id_buscado):
    low = 0
    high = len(lista_productos_ordenada_por_id) - 1

    while low <= high:
        mid = (low + high) // 2
        producto_medio = lista_productos_ordenada_por_id[mid]

        if producto_medio['id'] == id_buscado:
            return producto_medio
        elif producto_medio['id'] < id_buscado:
            low = mid + 1
        else:
            high = mid - 1

    return None
```

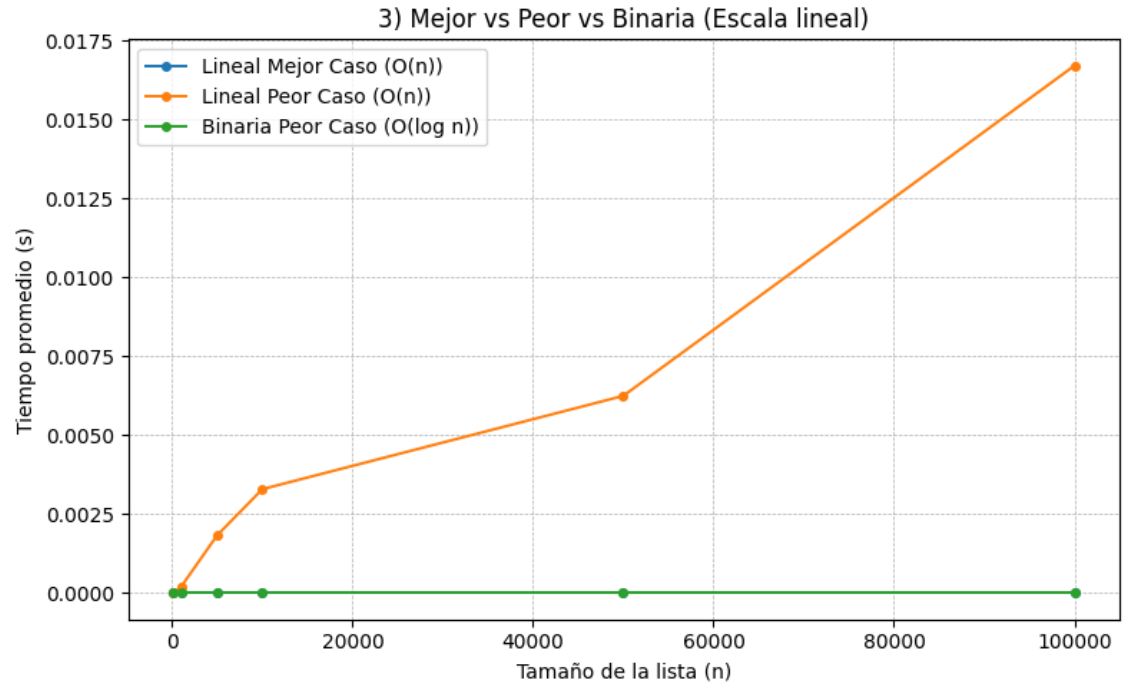
Anexo H.1 - Gráfico eficiencia búsqueda línea y binaria peor caso (escala logarítmica):



Anexo H.2 - Gráfico eficiencia búsqueda lineal y binaria peor caso (escala lineal):



Anexo H.3 - Gráfico eficiencia búsqueda lineal y binaria (mejor y pero caso lineal vs peor caso binaria):



Anexo H.4 - Gráfico búsqueda lineal y binaria (promedio):