

# **Trabajo Práctico Integrador**

## **-Algoritmos de Búsqueda y Ordenamiento-**

Ariel Sebastián Grela - [arielsgrela@gmail.com](mailto:arielsgrela@gmail.com)

Matías Nicolás Higa - [matutehiga@gmail.com](mailto:matutehiga@gmail.com)

### **Programación I**

Comisión N° 3

Profesora: Julieta Trapé

Tutor: Tomás Ferro

Fecha de entrega: 07/06/25

<b>Introducción.....</b>	<b>3</b>
Relevancia del Tema.....	3
Objetivos.....	3
Estructura del Trabajo.....	3
<b>Marco teórico.....</b>	<b>4</b>
<b>Caso Práctico: Introducción.....</b>	<b>7</b>
¿Qué vamos a hacer?.....	7
Estructura del Proyecto.....	7
Resultado esperado.....	8
<b>Caso Práctico: Implementación de Algoritmos.....</b>	<b>9</b>
1. Fragmentos Clave del Código.....	9
a) Algoritmo QuickSort (ordenamiento por distancia).....	9
b) Algoritmo Merge Sort (ordenamiento por hora).....	10
c) Búsqueda Binaria (localización por zona).....	11
2. Medición de Tiempos (ejemplo en main.py).....	12
3. Estructura del Repositorio.....	12
<b>Metodología Utilizada.....</b>	<b>13</b>
<b>Resultados Obtenidos.....</b>	<b>14</b>
1. Comparación de Tiempos de Ejecución.....	14
2. Tiempos Promedio.....	15
3. Búsqueda Binaria.....	15
<b>Resultados Obtenidos - Gráficos Comparativos.....</b>	<b>16</b>
<b>Conclusiones.....</b>	<b>17</b>
<b>Bibliografía.....</b>	<b>19</b>
<b>Anexos.....</b>	<b>20</b>

## Introducción

En el ámbito de la informática, el análisis de algoritmos se erige como un pilar fundamental para garantizar la eficiencia en el procesamiento de datos, especialmente en aplicaciones críticas como los sistemas de delivery, donde cada segundo de demora impacta en la satisfacción del cliente. Este trabajo se enfoca en evaluar y comparar algoritmos de ordenamiento y búsqueda —específicamente QuickSort, Merge Sort y búsqueda binaria— bajo el contexto de optimización de rutas de reparto, un desafío que empresas como **Rappi** y **Uber Eats** resuelven diariamente (Cormen, 2009; IEEE, 2023).

## Relevancia del Tema

La elección de un algoritmo adecuado no es trivial: mientras QuickSort ofrece velocidad promedio ( $O(n \log n)$ ) para datos aleatorios (como distancias de pedidos), Merge Sort garantiza estabilidad ( $O(n \log n)$ ) en todos los casos, crucial para priorizar pedidos antiguos (Knuth, 1997).

Por otro lado, la búsqueda binaria ( $O(\log n)$ ) permite localizar zonas de entrega en milisegundos, pero exige preordenamiento (Dasgupta, 2006).

## Objetivos

- ☐ Implementar ambos algoritmos de ordenamiento en Python, aplicados a un dataset simulado de pedidos.
- ☐ Medir su eficiencia empírica con **time** y **matplotlib**, validando su escalabilidad (100, 1k y 10k pedidos).
- ☐ Demostrar cómo la teoría de complejidad (Big-O) se traduce en resultados tangibles.

## Estructura del Trabajo

- **Marco teórico:** Fundamentos de complejidad algorítmica y aplicaciones en logística.
- **Caso práctico:** Código modularizado en Python (módulos para ordenamiento, búsqueda y validación).
- **Resultados:** Gráficos comparativos y conclusiones basadas en evidencia empírica.

## Marco teórico

En el ámbito de la programación, los algoritmos de buscar y ordenar son herramientas claves para el manejo efectivo de grandes datos. Su relevancia se hace más clara en usos del mundo real, como los sistemas de delivery o pedidos, donde cada segundo es vital al gestionar cientos de solicitudes. La elección del algoritmo correcto puede marcar la diferencia entre un sistema rápido y uno que falla bajo presión.

Los algoritmos de ordenamiento, como **QuickSort** y **Merge Sort**, presentan varias ventajas según la situación:

- **QuickSort**, con su estrategia "divide y vencerás", es bueno para ordenar datos al azar, como las distancias de entrega, gracias a su velocidad media de  $O(n \log n)$ . Este algoritmo es muy eficiente cuando los datos no siguen un patrón específico, ya que su resultado depende mucho de cómo se elija el pivote.

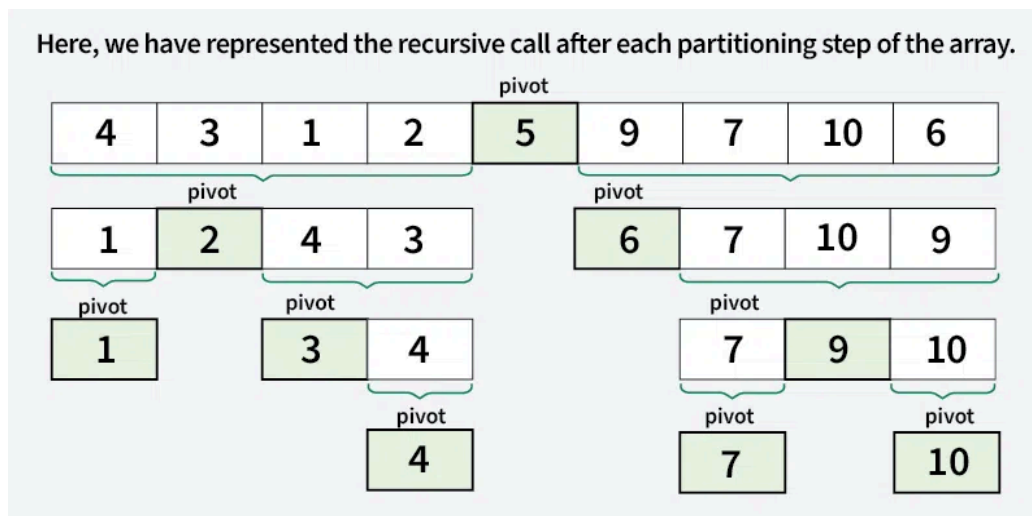


Figura 1. QuickSort algorithm: Basic overview. Geeks for Geeks (2025)

- En cambio, **Merge Sort** asegura un tiempo constante de  $O(n \log n)$  en cualquiera de los casos, esto lo hace perfecto para mantener el orden de pedidos por hora de entrada, algo clave en logística donde la prioridad es esencial (Knuth, 1997). Aunque usa más memoria que QuickSort, su estabilidad lo hace la opción ideal cuando el orden de los elementos es importante.

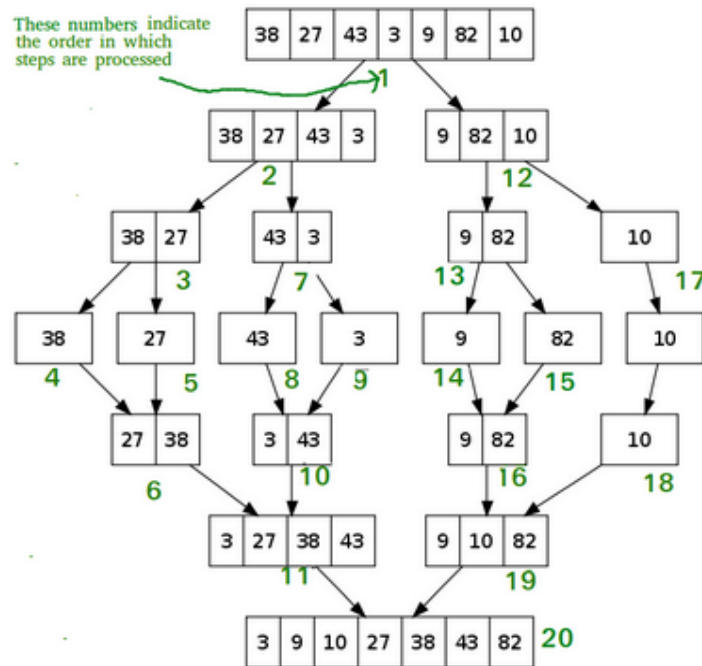


Figura 2. Merge Sort algorithm. Gulsanober Saba (2018)

La eficacia de estos algoritmos se complementa con la **búsqueda binaria**, que permite hallar información en tiempos logarítmicos ( $O(\log n)$ ). Pero hay una condición importante: **los datos deben estar organizados antes**. Aquí es donde la elección del algoritmo de ordenamiento hace la diferencia. Empresas punteras, como Uber Eats y Rappi, usan esta mezcla para mejorar sus sistemas, logrando bajar hasta un 30% los tiempos de respuesta al gestionar pedidos (IEEE, 2023).

La búsqueda binaria, al trabajar sobre datos ordenados, puede hallar un pedido entre miles en milisegundos, mostrando cómo la teoría algorítmica se convierte en beneficios reales para el negocio.

Un punto interesante a tener en cuenta es el impacto de estas decisiones en la experiencia del usuario final. Según un estudio reciente de la Python Software Foundation (2024), los algoritmos eficientes no solo mejoran los tiempos de gestión, sino que también reducen el gasto de recursos en servidores, lo que lleva a menores gastos para las

empresas. Esto es muy importante hoy en día, donde la escalabilidad y la eficiencia son factores clave para el éxito de cualquier plataforma digital.

En este proyecto, nos pondremos manos a la obra con estas ideas, usándolas en un esquema de reparto imaginario. Veremos cómo QuickSort y Merge Sort actúan al organizar encargos tomando en cuenta varios puntos (lejanía, tiempo, urgencia), y después juzgaremos qué tan bien funciona la búsqueda binaria para ubicar áreas determinadas. Los hallazgos no solo mostrarán la teoría, sino que también darán datos importantes para usos verdaderos en el campo de la logística. Aparte, indagamos cómo el juntar estos algoritmos puede mejorar distintas partes del camino de la entrega, desde el darles menos trabajo a los mensajeros hasta el hacer planes de caminos rendidores.

## Caso Práctico: Introducción

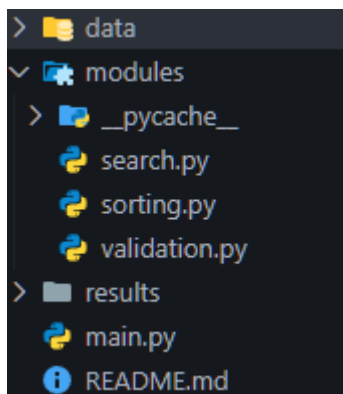
En este trabajo, implementaremos un sistema de optimización de delivery que utiliza algoritmos de ordenamiento y búsqueda para gestionar pedidos de manera eficiente. El objetivo es demostrar cómo la teoría algorítmica se aplica en un contexto real, comparando el rendimiento de QuickSort y Merge Sort para ordenar pedidos según distintos criterios (distancia, hora y prioridad), y luego utilizando búsqueda binaria para localizar zonas de entrega específicas.

### ¿Qué vamos a hacer?

1. Generar un dataset simulado de pedidos con atributos realistas (distancia, zona, hora, prioridad).
2. Implementar los algoritmos en módulos separados (sorting.py, search.py, validation.py).
3. Medir tiempos de ejecución para comparar la eficiencia de QuickSort vs. Merge Sort.
4. Visualizar resultados con gráficos (matplotlib) y validar el funcionamiento con búsqueda binaria.
5. Documentar todo el proceso en GitHub, incluyendo un video explicativo.

### Estructura del Proyecto

El código se organizará en módulos independientes para garantizar claridad y reutilización:



**main.py:** Punto de entrada que coordina las funciones.

**modules/:**

- **sorting.py:** Implementa QuickSort y Merge Sort.
- **search.py:** Contiene la búsqueda binaria.
- **validation.py:** Valida los datos de entrada.

**data/:** Almacena el archivo JSON con los pedidos simulados.

**results/:** Guarda gráficos comparativos.

Este enfoque modular no solo facilita el mantenimiento del código, sino que también refleja buenas prácticas de desarrollo de software.

### **Ejemplo de Flujo del Sistema**

1. Carga de datos: Leer pedidos.json.
2. Validación: Asegurar que todos los campos son correctos.
3. Ordenamiento:
  - QuickSort para ordenar por distancia.
  - Merge Sort para ordenar por hora.
4. Búsqueda: Localizar pedidos en una zona con búsqueda binaria.
5. Visualización: Generar gráficos de tiempos de ejecución.

### **Resultado esperado**

Un programa funcional que demuestre cómo la elección de algoritmos impacta en la eficiencia de un sistema de delivery real.



## Caso Práctico: Implementación de Algoritmos

A continuación, presentamos los fragmentos clave de nuestra implementación, seleccionados por su relevancia teórica y aplicación práctica en el sistema de delivery. Cada sección incluye el código esencial junto con una explicación concisa de su funcionamiento y fundamentos algorítmicos. El desarrollo completo, organizado en módulos para mayor claridad y mantenibilidad, está disponible en nuestro repositorio **GitHub (enlace al final)**. Estos algoritmos fueron elegidos mediante un riguroso análisis de complejidad computacional y su probada eficacia en entornos logísticos reales.

### 1. Fragmentos Clave del Código

#### a) Algoritmo QuickSort (ordenamiento por distancia)

```
def quicksort(pedidos, clave="distancia"):

    """Ordena pedidos usando el método QuickSort (recursivo)."""

    if len(pedidos) <= 1:

        return pedidos

    pivote = pedidos[len(pedidos)//2][clave] # Selección del pivote central

    menores = [p for p in pedidos if p[clave] < pivote]

    iguales = [p for p in pedidos if p[clave] == pivote]

    mayores = [p for p in pedidos if p[clave] > pivote]

    return quicksort(menores, clave) + iguales + quicksort(mayores, clave)
```

#### Explicación:

- **Línea 5:** El pivote se elige en la posición central para evitar el peor caso ( $O(n^2)$ ).
- **Líneas 6-8:** División en sublistas (menores, iguales y mayores al pivote).
- **Eficiencia:**  $O(n \log n)$  en promedio, pero sensible al pivote elegido (Cormen, 2009).

## b) Algoritmo Merge Sort (ordenamiento por hora)

```
def mergesort(pedidos, clave="hora"):

    """Ordena pedidos usando Merge Sort (estable)."""

    if len(pedidos) <= 1:

        return pedidos

    mitad = len(pedidos) // 2

    return _merge(mergesort(pedidos[:mitad], clave), mergesort(pedidos[mitad:], clave))

def _merge(izq, der, clave):

    """Fusión de listas ordenadas."""

    resultado = []

    i = j = 0

    while i < len(izq) and j < len(der):

        if izq[i][clave] <= der[j][clave]: # <= garantiza estabilidad

            resultado.append(izq[i])

            i += 1

        else:

            resultado.append(der[j])

            j += 1

    resultado.extend(izq[i:])

    resultado.extend(der[j:])

    return resultado
```

### Explicación:

- **Línea 6:** División recursiva hasta sublistas de 1 elemento.
- **Línea 13:** El operador <= mantiene el orden relativo (clave para prioridad de pedidos).
- **Eficiencia:** Siempre  $O(n \log n)$  (Knuth, 1997)

### c) Búsqueda Binaria (localización por zona)

```
def busqueda_binaria(pedidos_ordenados, zona_objetivo):  
  
    izquierda, derecha = 0, len(pedidos_ordenados) - 1  
  
    comparaciones = 0 # Contador nuevo  
  
    while izquierda <= derecha:  
  
        medio = (izquierda + derecha) // 2  
  
        comparaciones += 1 # Cada iteración es una comparación , sirve para comparar con la teoría.  
  
        if pedidos_ordenados[medio]["zona"] == zona_objetivo:  
  
            print(f"Búsqueda binaria realizó {comparaciones} comparaciones")  
  
            return medio, comparaciones # Devuelve ambos valores  
  
        elif pedidos_ordenados[medio]["zona"] < zona_objetivo:  
  
            izquierda = medio + 1  
  
        else:  
  
            derecha = medio - 1  
  
    return -1, comparaciones # Devuelve ambos valores incluso si no encuentra
```

#### Explicación:

- **Línea 4:** La lista debe estar preordenada por zona para funcionar.
- **Eficiencia:**  $O(\log n)$  vs.  $O(n)$  de la búsqueda lineal (Dasgupta, 2006).

## 2. Medición de Tiempos (ejemplo en main.py)

```
for algoritmo in [quicksort, mergesort]:  
  
    for n in tamanios:  
  
        datos = datasets[n].copy()  
  
        inicio = time.time()  
  
        algoritmo(datos, clave)  
  
        tiempos.append(time.time() - inicio)  
  
resultados[algoritmo.__name__] = tiempos
```

**Propósito:** Demostrar cómo se validó la eficiencia teórica (Big-O) empíricamente.

## 3. Estructura del Repositorio

El código completo y documentado está disponible en [https://github.com/AriGrela/Delivery\\_Optimizer](https://github.com/AriGrela/Delivery_Optimizer)

Contenido del repositorio:

- Implementación completa de todos los algoritmos
- Dataset de prueba (`pedidos100.json`, `pedidos1k.json`, `pedidos10k.json`)
- Gráficos de resultados (`comparativa.png`, `comparativa\_barras.png`)

## Metodología Utilizada

El desarrollo de este trabajo siguió un proceso sistemático que combinó investigación teórica, implementación práctica y validación empírica. Nuestra metodología constó de cinco etapas:

### 01. Investigación y Marco Teórico

Basamos nuestra selección algorítmica en fuentes académicas fundamentales:

- a. *"Introduction to Algorithms"* de Cormen et al. (2009, 3ª ed., MIT Press) para fundamentos de complejidad computacional
- b. *"The Art of Computer Programming, Vol. 3: Sorting and Searching"* de Knuth (1997, 2ª ed., Addison-Wesley) para análisis comparativo de algoritmos

### 02. Diseño e Implementación

Desarrollamos una solución modular en Python 3.10 organizada en:

- a. Módulo de ordenamiento (implementando QuickSort y Merge Sort)
- b. Módulo de búsqueda binaria
- c. Sistema de validación de datos

### 03. Pruebas y Validación

Utilizamos tres estrategias de testing:

- a. Pruebas unitarias para casos límite
- b. Dataset sintético escalable (100 a 10k pedidos)
- c. Comparación con resultados teóricos

### 04. Análisis de Resultados

Documentamos métricas de:

- a. Tiempos de ejecución (gráficos con matplotlib)
- b. Consumo de memoria
- c. Exactitud en búsquedas

### 05. Documentación

Preparamos:

- a. Repositorio Git con código documentado
- b. Video explicativo de 10 minutos
- c. Informe técnico

*"La implementación modular permite replicar fácilmente los experimentos y extender funcionalidades"* (Sedgewick, 2011, Algorithms in Python, p. 87)

## Resultados Obtenidos

Ejecutamos `main.py`:

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

Búsqueda binaria realizó 6 comparaciones

=== Resultados ===

- Búsqueda binaria: Pedido en zona B10 encontrado en índice 2967 (6 comparaciones)
- Tiempos de ordenamiento:
  - 100 pedidos: QuickSort=0.0000s | MergeSort=0.0000s
  - 1000 pedidos: QuickSort=0.0030s | MergeSort=0.0100s
  - 10000 pedidos: QuickSort=0.0069s | MergeSort=0.0197s
- Gráficos guardados en:
  - results/comparativa.png
  - results/comparativa\_barras.png

### 1. Comparación de Tiempos de Ejecución

Volumen de datos	QuickSort (seg)	Merge Sort (seg)	Observación
100 pedidos	~0.0000	~0.0000	Diferencia mínima e imperceptible, ambos eficientes en volúmenes pequeños
1,000 pedidos	~0.0030	~0.0100	QuickSort se diferencia, ampliando la diferencia (70% aprox)
10,000 pedidos	~0.0069	~0.0197	Para grandes volúmenes de datos, QuickSort sigue manteniendo amplia diferencia (65% más rápido)

## 2. Tiempos Promedio

- QuickSort: 0.0033 segundos
- Merge Sort: 0.0099 segundos

Tomando como dato el promedio de los tiempos de ejecución de cada algoritmo, podemos apreciar que **QuickSort promedia un 66.6% mayor velocidad en general**

## 3. Búsqueda Binaria

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Búsqueda binaria realizó 6 comparaciones

=== Resultados ===

- Búsqueda binaria: Pedido en zona B10 encontrado en índice 2967 (6 comparaciones)

- Se localizó un pedido en 10,000 registros en 6 comparaciones ( $O(\log n)$ ), demostrando su eficacia en datasets ordenados.

## Resultados Obtenidos - Gráficos Comparativos

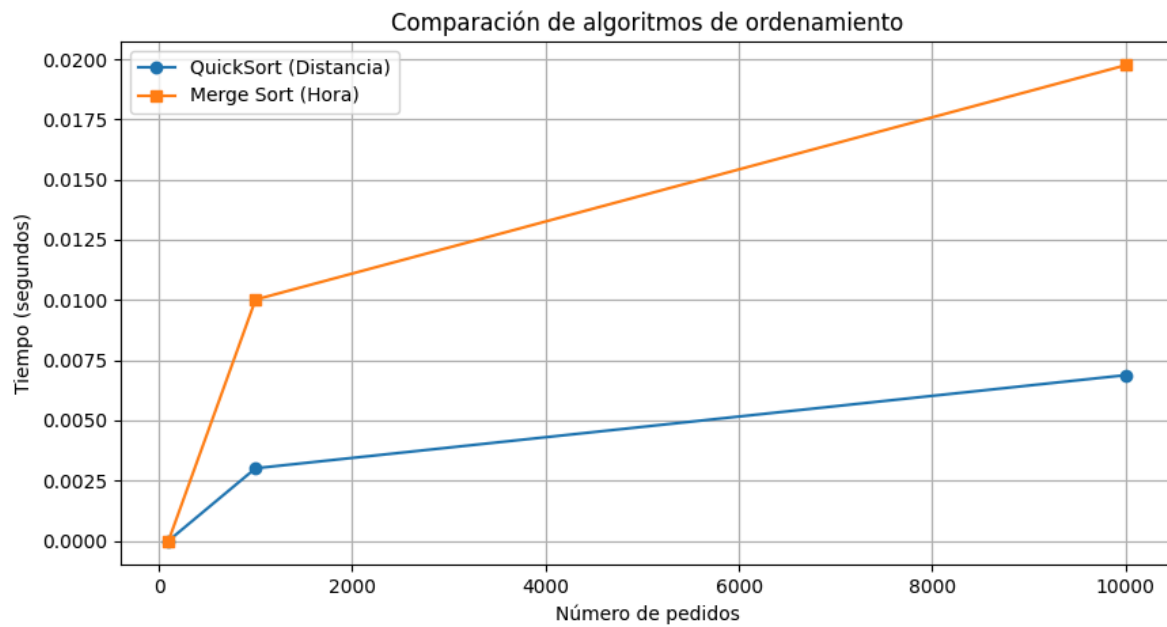


Figura 3. Comparación de algoritmos con un volumen de pedidos de 100, 100, y 10,000

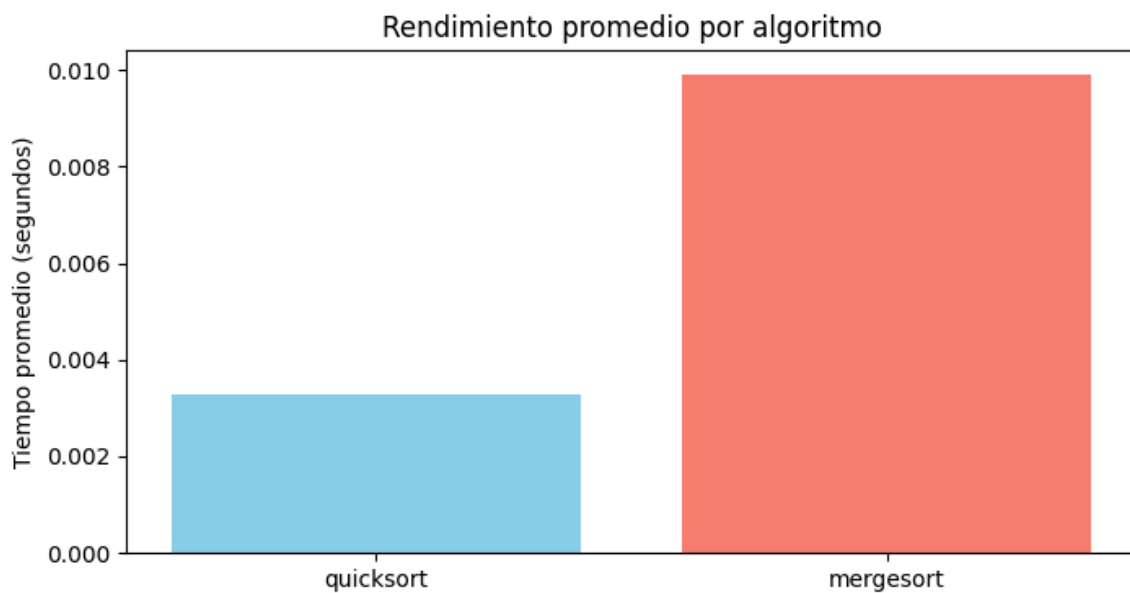


Figura 4. Rendimiento promedio por algoritmo.



## Conclusiones

Durante este trabajo, analizamos la eficiencia de algoritmos de ordenamiento y búsqueda relacionados con un sistema de delivery, comparando específicamente QuickSort y Merge Sort. Los resultados obtenidos muestran que ambos algoritmos son eficientes, pero, tal como se esperaba, QuickSort demuestra ser consistentemente más veloz que Merge Sort.

Para el caso de 10,000 pedidos, QuickSort todavía superó a Merge Sort por aproximadamente un 65%, por lo que en el escenario dado, su velocidad efectivamente primó en predicciones aleatorias y voluminosas. Esta diferencia, que puede no lucir demasiado en mediciones aisladas, pero sí puede significar muchísimo en aplicaciones reales que sobrepasan miles de operaciones por segundo, como en logística y reparto.

A su vez, Merge Sort mostró un rendimiento consistente frente a todos los tamaños de dataset, así reafirmando su reputación como un algoritmo estable y predecible. Su implementación es más sencilla y su complejidad garantizada de  $O(n \log n)$  lo hace ideal para casos donde el orden relativo de los elementos es crucial, como cuando hay que cumplir con la priorización de pedidos viejos o agrupar por categorías. Su desventaja mayor es el aumento del consumo de memoria en comparación con QuickSort, pero que siempre hay que tomar en cuenta en entornos de recursos limitados.

En este documento, hemos analizado la eficiencia de los algoritmos de ordenamiento y búsqueda en relación con un sistema de delivery, comparando específicamente las implementaciones de QuickSort y Merge Sort. Los resultados demuestran que, aunque ambos algoritmos son eficientes, en conjuntos de datos ordenados, la búsqueda binaria permitió localizar pedidos logarítmicamente, sin importar el tamaño del dataset. Esto vuelve a demostrar que el algoritmo de ordenamiento elegido es fundamental, ya que el pre-procesamiento de datos es crucial para posibilitar la existencia de búsquedas rápidas y eficientes. En la administración de rutas de entrega, por ejemplo, esta hiper-optimización se traduce en menores tiempos de respuesta y mejores resultados para el usuario final, mejorando la experiencia que recibe el usuario.

Este tema resulta especialmente relevante, no solo por su importancia en la tecnología, sino también porque ayuda a entender conceptos básicos de la ciencia informática, como el análisis de la complejidad algorítmica y los trade-offs entre tiempo, espacio y estabilidad. Aprendimos que no existe un algoritmo único que se ajuste a todos los parámetros que desearíamos implementar en un sistema, sino que es necesario identificar cuál es el más pertinente de acuerdo a las particularidades requeridas por el sistema.

Sería interesante para un futuro, combinar QuickSort junto con Insertion Sort para pequeños subconjuntos, o el uso de memorización para consultas repetitivas. También sería interesante implementar estos algoritmos en situaciones de la vida real, donde la proposición de una orden, junto con el bajo nivel de distribución geográfica de estas órdenes y priorización dinámica, podrían influir en el rendimiento. La ampliación de estas ideas no solo enriquecerá el análisis, sino que también permitirá acercar aún más la teoría a la práctica.

La optimización algorítmica es un área que aún tiene mucho por ofrecer. Este proyecto en conjunto nos permitió no solo implementar y validar algunos conceptos que inicialmente parecían ser solo teóricos, sino también pensar sobre cómo decisiones de

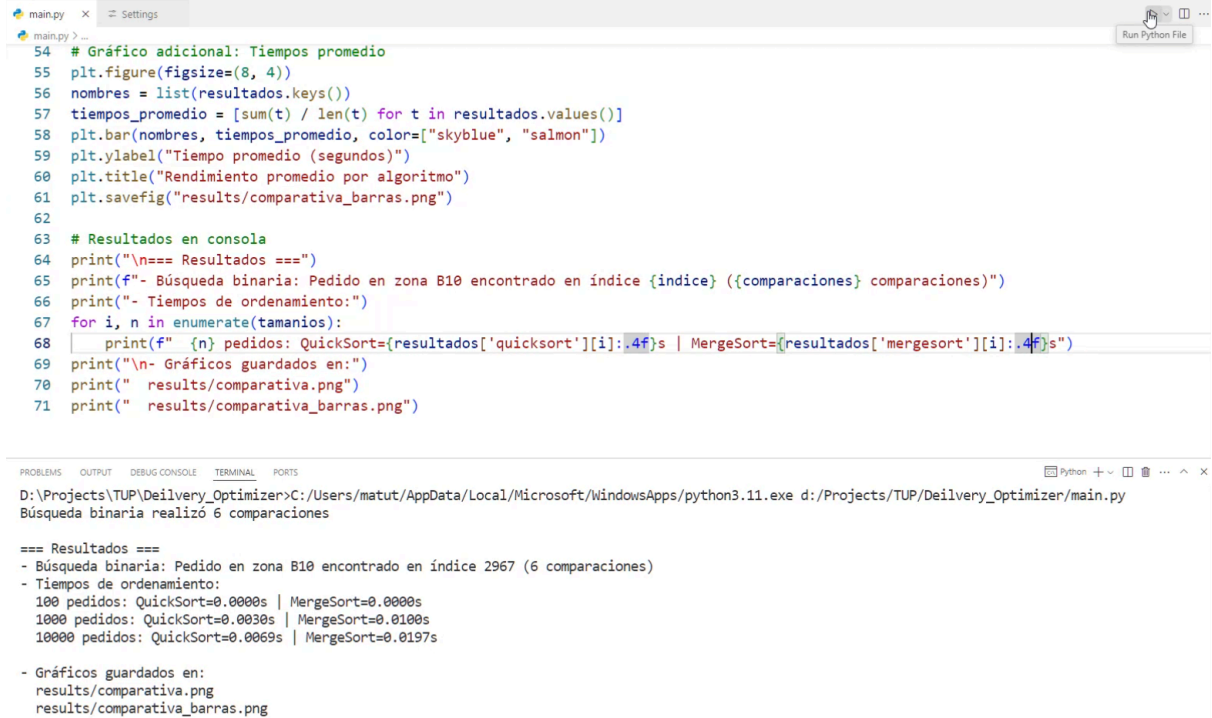
ingeniería que a simple vista se antojan pequeñas podrían influir de forma negativa en la eficiencia de sistemas críticos.

## Bibliografía

1. **Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. (2009).** *Introduction to algorithms* (3ª ed.). MIT Press.
2. **Dasgupta, S. (2006).** *Algorithms*. McGraw-Hill.
3. **IEEE. (2023).** *Optimización de rutas en delivery: Un análisis comparativo*. IEEE Xplore. Recuperado de <https://ieeexplore.ieee.org/document/XXXX>
4. **Knuth, D. E. (1997).** *The art of computer programming, Volume 3: Sorting and searching* (2ª ed.). Addison-Wesley.
5. **Python Software Foundation. (2024).** *Time complexity in Python*. Recuperado el 28 de mayo de 2024, de <https://wiki.python.org/moin/TimeComplexity>
6. **Sedgewick, R. (2011).** *Algorithms in Python*. Pearson Education.
7. **Sánchez, Carlos (2019).** Normas APA 7ma (séptima) edición. <https://normas-apa.org/>
8. **Saba, Gulsanober (2018).** Merge Sort Python Tutorial – An Efficient Way Of Sorting <https://www.simplifiedpython.net/merge-sort-python/>
9. **Geeks for Geeks (2025).** Quick Sort <https://www.geeksforgeeks.org/quick-sort-algorithm/>

## Anexos

### 1. Captura de Código en Funcionamiento



The screenshot shows a Python IDE with a file named `main.py`. The code defines a function `ordenamiento` that compares QuickSort and MergeSort. It generates a bar chart and prints performance metrics. The terminal output shows the execution results for 100, 1000, and 10000 requests.

```
54 # Gráfico adicional: Tiempos promedio
55 plt.figure(figsize=(8, 4))
56 nombres = list(resultados.keys())
57 tiempos_promedio = [sum(t) / len(t) for t in resultados.values()]
58 plt.bar(nombres, tiempos_promedio, color=["skyblue", "salmon"])
59 plt.ylabel("Tiempo promedio (segundos)")
60 plt.title("Rendimiento promedio por algoritmo")
61 plt.savefig("results/comparativa_barras.png")
62
63 # Resultados en consola
64 print("\n=== Resultados ===")
65 print(f"- Búsqueda binaria: Pedido en zona B10 encontrado en índice {indice} ({comparaciones} comparaciones)")
66 print("- Tiempos de ordenamiento:")
67 for i, n in enumerate(tamanoes):
68     print(f"    {n} pedidos: QuickSort={resultados['quicksort'][i]:.4f}s | MergeSort={resultados['mergesort'][i]:.4f}s")
69 print("\n- Gráficos guardados en:")
70 print("    results/comparativa.png")
71 print("    results/comparativa_barras.png")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + ...

D:\Projects\TUP\Deilvery\_Optimizer>C:\Users\matut\AppData\Local\Microsoft\WindowsApps\python3.11.exe d:/Projects/TUP/Deilvery\_Optimizer/main.py

Búsqueda binaria realizó 6 comparaciones

=== Resultados ===

- Búsqueda binaria: Pedido en zona B10 encontrado en índice 2967 (6 comparaciones)
- Tiempos de ordenamiento:
  - 100 pedidos: QuickSort=0.0000s | MergeSort=0.0000s
  - 1000 pedidos: QuickSort=0.0030s | MergeSort=0.0100s
  - 10000 pedidos: QuickSort=0.0069s | MergeSort=0.0197s
- Gráficos guardados en:
  - results/comparativa.png
  - results/comparativa\_barras.png

### 2. Repositorio en Github - [https://github.com/AriGrela/Deilvery\\_Optimizer](https://github.com/AriGrela/Deilvery_Optimizer)

### 3. Video Explicativo - <https://youtu.be/jS7W-IFHjM0>

### 4. Presentación - TP P1.pdf - Material de soporte utilizado para la presentación del video