

# Trabajo Práctico Integrador

## Algoritmos de Búsqueda y Ordenamiento

### Programación 1

**Profesora**  
Julieta Trapé

Comisión 3

**Tutor**  
Tomás Ferro

# Integrantes

- Ariel Grela
- Matías Higa

# Introducción

---

# Introducción

Este estudio compara algoritmos de ordenamiento (QuickSort, Merge Sort) y búsqueda (búsqueda binaria) en el contexto de optimización de rutas de reparto, clave en plataformas como *Rappi* o Uber Eats

## Relevancia

- QuickSort: rápido con datos aleatorios ( $O(n \log n)$ )
- Merge Sort: estable en todos los casos, ideal para priorización
- Búsqueda Binaria: alta eficiencia ( $O(\log n)$ ) con datos ordenados

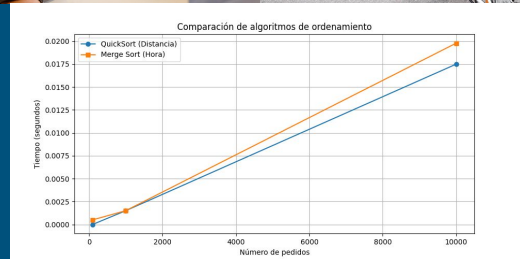


# Introducción

Este estudio compara algoritmos de ordenamiento (QuickSort, Merge Sort) y búsqueda (búsqueda binaria) en el contexto de optimización de rutas de reparto, clave en plataformas como *Rappi* o *Uber Eats*

## Objetivos

- Implementar los algoritmos en Python
- Medir eficiencia con datasets simulados (100, 1k, 10k pedidos)
- Validar la teoría de complejidad con resultados reales



# Marco Teórico

---

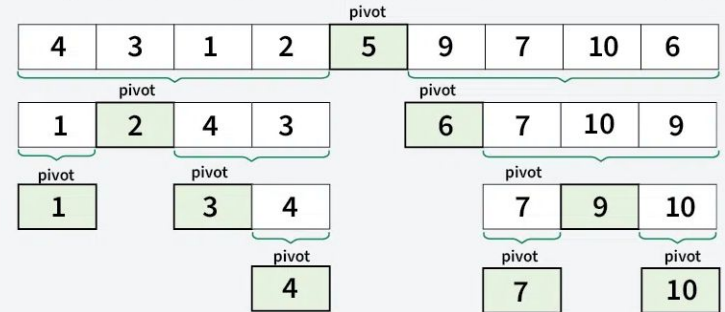
# Marco Teórico

## QuickSort

Ordenamiento eficiente para datos aleatorios:

- Usa la estrategia divide y vencerás
- Muy rápido en promedio:  $O(n \log n)$
- Ideal para datos sin patrón (como distancias de entrega)
- Su rendimiento depende de la elección del pivote
- Puede volverse lento ( $O(n^2)$ ) si el pivote no se elige bien

Here, we have represented the recursive call after each partitioning step of the array.

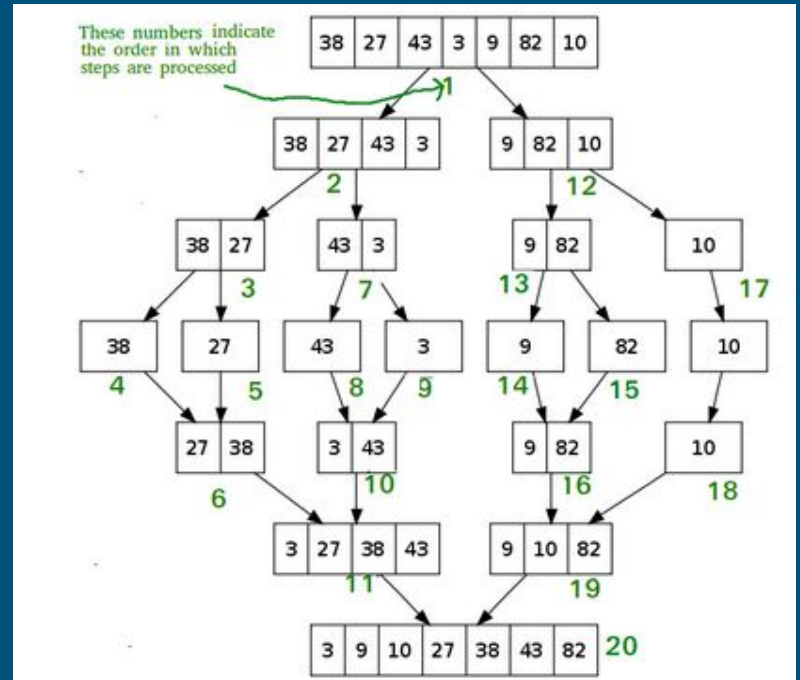


# Marco Teórico

## Merge Sort

Orden estable y rendimiento garantizado:

- Siempre  $O(n \log n)$ , incluso en el peor caso
- Mantiene el orden original de los elementos (estable)
- Ideal para pedidos por hora de entrada o prioridades
- Usa más memoria que QuickSort
- Perfecto cuando el orden es clave





# Marco Teórico

---

## Búsqueda Binaria

Precisión rápida en datos ordenados:

- Tiempo de búsqueda:  $O(\log n)$
- Requiere que los datos estén ordenados previamente
- Clave en apps como Uber Eats y Rappi
- Reduce tiempos de respuesta hasta un 30%
- Encuentra un pedido entre miles en milisegundos

# Caso Práctico

---

# Caso Práctico: Introducción

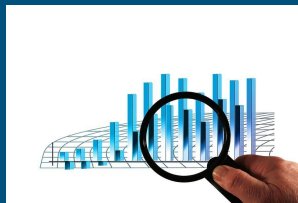
## Introducción

Optimización de pedidos con algoritmos

- Aplicamos teoría algorítmica en un caso real de delivery
- Ordenamos pedidos por distancia, hora y prioridad
- Simulamos un dataset
- Comparamos QuickSort vs. Merge Sort
- Usamos búsqueda binaria para localizar zonas
- Implementación modular: `sorting.py`, `search.py`, `validation.py`
- Visualización con gráficos (matplotlib)
- Documentación en GitHub

*Rappi*

**GitHub**

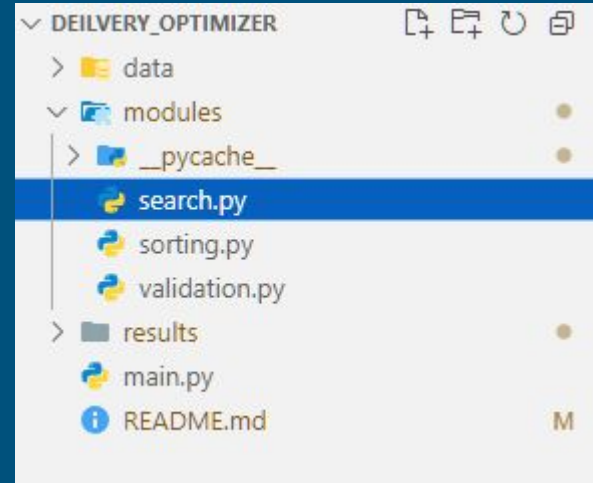


# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Búsqueda Binaria

 modules / search.py



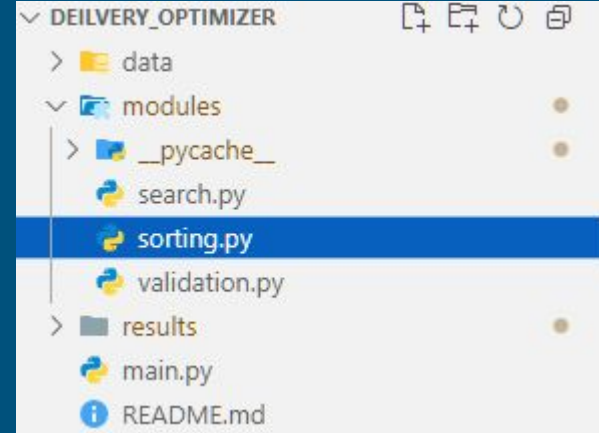
# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

 modules / sorting.py

Algoritmos de ordenamiento:

- Quicksort
- Merge Sort



# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Algoritmo QuickSort



modules / search.py

- Ordenamiento por **distancia** de pedido

modules > sorting.py > mergesort

```
1 def quicksort(pedidos, clave="distancia"):
2     """Ordena pedidos usando QuickSort (in-place)."""
3     if len(pedidos) <= 1:
4         return pedidos
5     pivot = pedidos[len(pedidos)//2][clave]
6     menores = [p for p in pedidos if p[clave] < pivot]
7     iguales = [p for p in pedidos if p[clave] == pivot]
8     mayores = [p for p in pedidos if p[clave] > pivot]
9     return quicksort(menores, clave) + iguales + quicksort(mayores, clave)
```

# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Algoritmo QuickSort

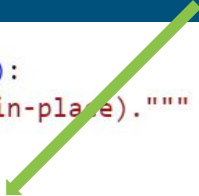


modules / search.py

- Ordenamiento por **distancia** de pedido
- El pivote se elige en la posición central

modules > sorting.py > mergesort

```
1 def quicksort(pedidos, clave="distancia"):
2     """Ordena pedidos usando QuickSort (in-place)."""
3     if len(pedidos) <= 1:
4         return pedidos
5     pivot = pedidos[len(pedidos)//2][clave]
6     menores = [p for p in pedidos if p[clave] < pivot]
7     iguales = [p for p in pedidos if p[clave] == pivot]
8     mayores = [p for p in pedidos if p[clave] > pivot]
9     return quicksort(menores, clave) + iguales + quicksort(mayores, clave)
```



# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Algoritmo QuickSort



modules / search.py

- Ordenamiento por **distancia** de pedido
- El pivote se elige en la posición central
- Ayuda a dividir los elementos en menores, iguales y mayores

```
modules > sorting.py > mergesort
1 def quicksort(pedidos, clave="distancia"):
2     """Ordena pedidos usando QuickSort (in-place)."""
3     if len(pedidos) <= 1:
4         return pedidos
5     pivot = pedidos[len(pedidos)//2][clave]
6     menores = [p for p in pedidos if p[clave] < pivot]
7     iguales = [p for p in pedidos if p[clave] == pivot]
8     mayores = [p for p in pedidos if p[clave] > pivot]
9     return quicksort(menores, clave) + iguales + quicksort(mayores, clave)
```



# Caso Práctico: Implementación de Algoritmos


## Fragmentos clave del código

### Algoritmo Merge Sort



modules / search.py

- Ordenamiento por **hora** de pedido



```
11 def mergesort(pedidos, clave="hora"):
12     """Ordena pedidos usando Merge Sort (estable)."""
13     if len(pedidos) <= 1:
14         return pedidos
15     mitad = len(pedidos) // 2
16     izquierda = mergesort(pedidos[:mitad], clave)
17     derecha = mergesort(pedidos[mitad:], clave)
18     return _merge(izquierda, derecha, clave)
```

```
20 def _merge(izq, der, clave):
21     """Fusión de listas ordenadas."""
22     resultado = []
23     i = j = 0
24     while i < len(izq) and j < len(der):
25         if izq[i][clave] <= der[j][clave]:
26             resultado.append(izq[i])
27             i += 1
28         else:
29             resultado.append(der[j])
30             j += 1
31     resultado.extend(izq[i:])
32     resultado.extend(der[j:])
33     return resultado
```

# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Algoritmo Merge Sort



modules / search.py

- Ordenamiento por **hora** de pedido
- Divide la lista en mitades hasta que tiene listas de tamaño 1.

```
11 def mergesort(pedidos, clave="hora"):
12     """Ordena pedidos usando Merge Sort (estable)."""
13     if len(pedidos) <= 1:
14         return pedidos
15     mitad = len(pedidos) // 2
16     izquierda = mergesort(pedidos[:mitad], clave)
17     derecha = mergesort(pedidos[mitad:], clave)
18     return merge(izquierda, derecha, clave)
```

```
20 def _merge(izq, der, clave):
21     """Fusión de listas ordenadas."""
22     resultado = []
23     i = j = 0
24     while i < len(izq) and j < len(der):
25         if izq[i][clave] <= der[j][clave]:
26             resultado.append(izq[i])
27             i += 1
28         else:
29             resultado.append(der[j])
30             j += 1
31     resultado.extend(izq[i:])
32     resultado.extend(der[j:])
33     return resultado
```

# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Algoritmo Merge Sort



modules / search.py

- Ordenamiento por **hora** de pedido
- Divide la lista en mitades hasta que tiene listas de tamaño 1
- Combina las listas ordenadas con `_merge`

```
11 def mergesort(pedidos, clave="hora"):
12     """Ordena pedidos usando Merge Sort (estable)."""
13     if len(pedidos) <= 1:
14         return pedidos
15     mitad = len(pedidos) // 2
16     izquierda = mergesort(pedidos[:mitad], clave)
17     derecha = mergesort(pedidos[mitad:], clave)
18     return _merge(izquierda, derecha, clave)
```

```
20 def _merge(izq, der, clave):
21     """Fusión de listas ordenadas."""
22     resultado = []
23     i = j = 0
24     while i < len(izq) and j < len(der):
25         if izq[i][clave] <= der[j][clave]:
26             resultado.append(izq[i])
27             i += 1
28         else:
29             resultado.append(der[j])
30             j += 1
31     resultado.extend(izq[i:])
32     resultado.extend(der[j:])
33     return resultado
```

# Caso Práctico: Implementación de Algoritmos

## Medición de Tiempos



main.py

Comparar el tiempo de ejecución

Quicksort (ordenando por "distancia")

Merge Sort (ordenando por "hora")

para datasets de distintos tamaños: 100, 1000 y 10000 pedidos.

```
25 # Medir tiempos de ordenamiento
26 resultados = {}
27 for algoritmo in [quicksort, mergesort]:
28     tiempos = []
29     for n in tamanios:
30         datos = datasets[n].copy() # Usamos copia para no modificar el original
31         inicio = time.time()
32         algoritmo(datos, "distancia" if algoritmo == quicksort else "hora")
33         tiempos.append(time.time() - inicio)
34     resultados[algoritmo.__name__] = tiempos
```

# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Búsqueda Binaria



main.py

Líneas 36 a 38 #

```
main.py x search.py
main.py > ...
36 # Búsqueda binaria (usamos el dataset de 10k como ejemplo y cuenta cantidad de comparaciones.)
37 pedidos_por_zona = sorted(datasets[10000], key=lambda x: x["zona"])
38 indice, comparaciones = busqueda_binaria(pedidos_por_zona, "B10")
39
```

Dataset utilizado: 10000 pedidos

Localización por zona **B10**

# Caso Práctico: Implementación de Algoritmos

## Fragmentos clave del código

### Búsqueda Binaria

 main.py

Líneas 36 a 38 #

```
main.py x search.py  
main.py > ...  
36 # Búsqueda binaria (usamos el dataset de 10k como ejemplo y cuenta cantidad de comparaciones.)  
37 pedidos_por_zona = sorted(datasets[10000], key=lambda x: x["zona"])  
38 indice, comparaciones = busqueda_binaria(pedidos_por_zona, "B10")  
39
```

Localización por zona **B10**

# Resultados Obtenidos

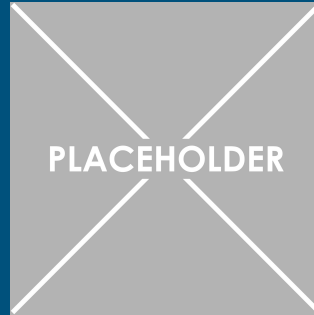
---

# Resultados Obtenidos

---



main.py



video corriendo el script



# Resultados Obtenidos

## Resultados de ejecución de 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

# Resultados Obtenidos

## Comparación de tiempos de ejecución

- 3 sets de datos (100 pedidos, 1000 pedidos, 10000 pedidos)
- QuickSort y Merge Sort

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)

# Resultados Obtenidos

## Comparación de tiempos de ejecución

- 3 sets de datos (100 pedidos, 1000 pedidos, 10000 pedidos)
- QuickSort y Merge Sort

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)

# Resultados Obtenidos

## Comparación de tiempos de ejecución

- 3 sets de datos (100 pedidos, 1000 pedidos, 10000 pedidos)
- QuickSort y Merge Sort

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)

# Resultados Obtenidos

## Comparación de tiempos de ejecución

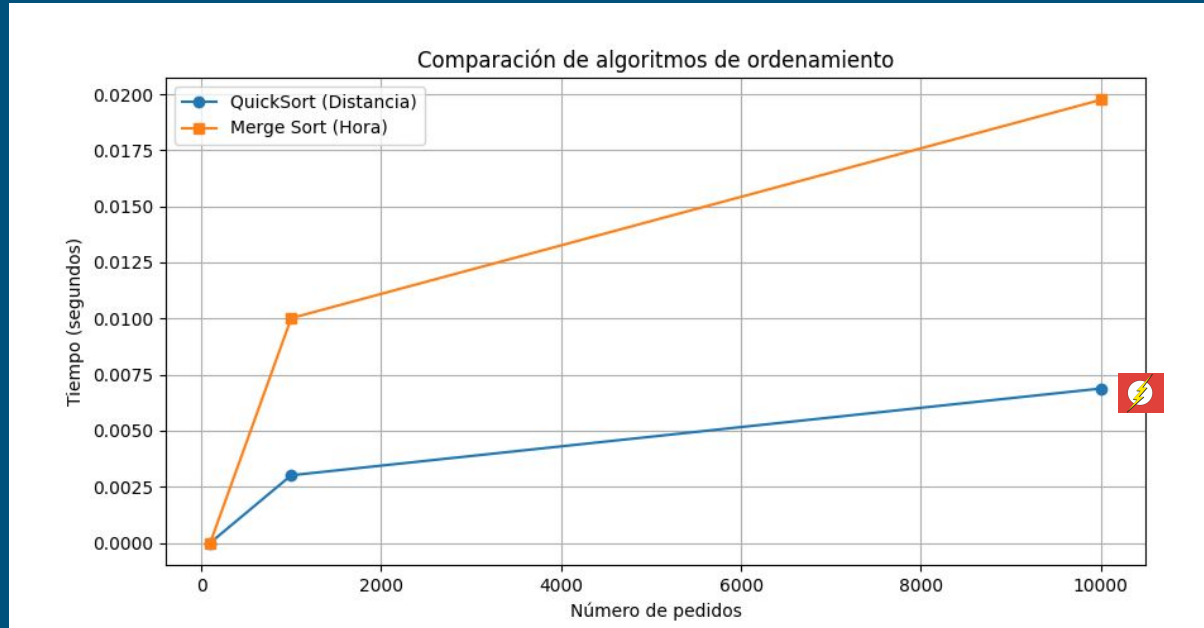
- 3 sets de datos (100 pedidos, 1000 pedidos, 10000 pedidos)
- QuickSort y Merge Sort

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)

# Resultados Obtenidos

## Gráficos comparativos

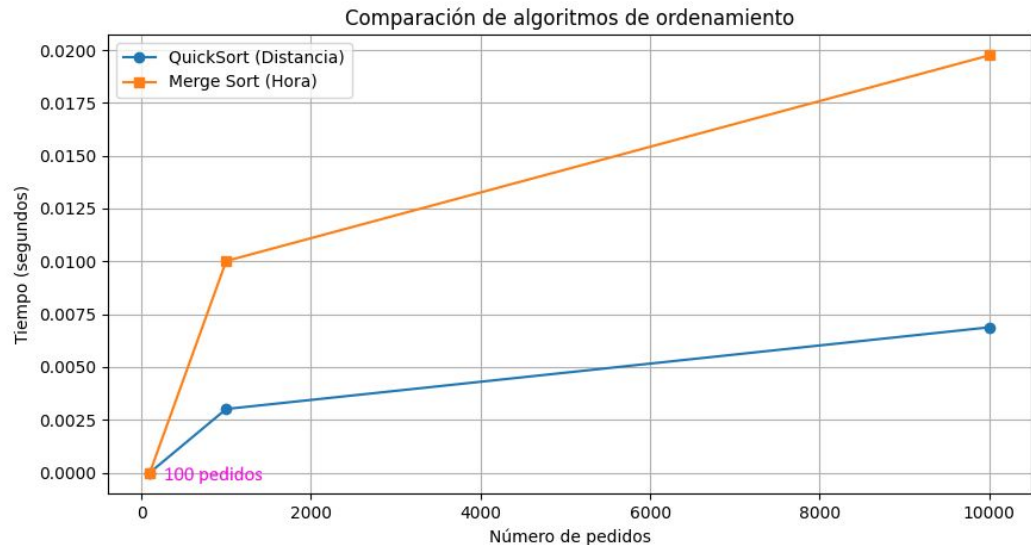
- QuickSort: 0.0033 seg
- Merge Sort: 0.099 seg



# Resultados Obtenidos

## Gráficos comparativos

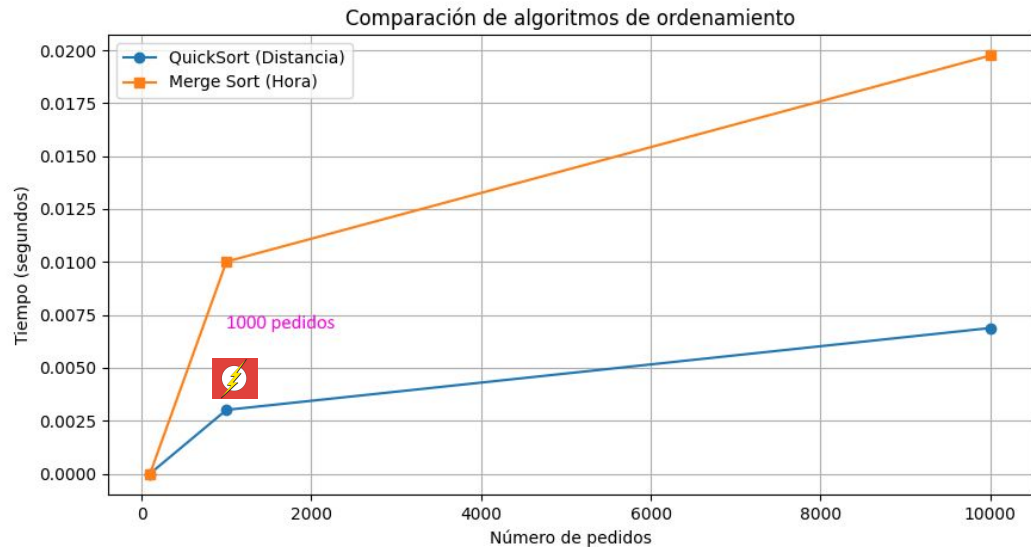
- QuickSort: 0.0033 seg
- Merge Sort: 0.099 seg



# Resultados Obtenidos

## Gráficos comparativos

- QuickSort: 0.0033 seg
- Merge Sort: 0.099 seg





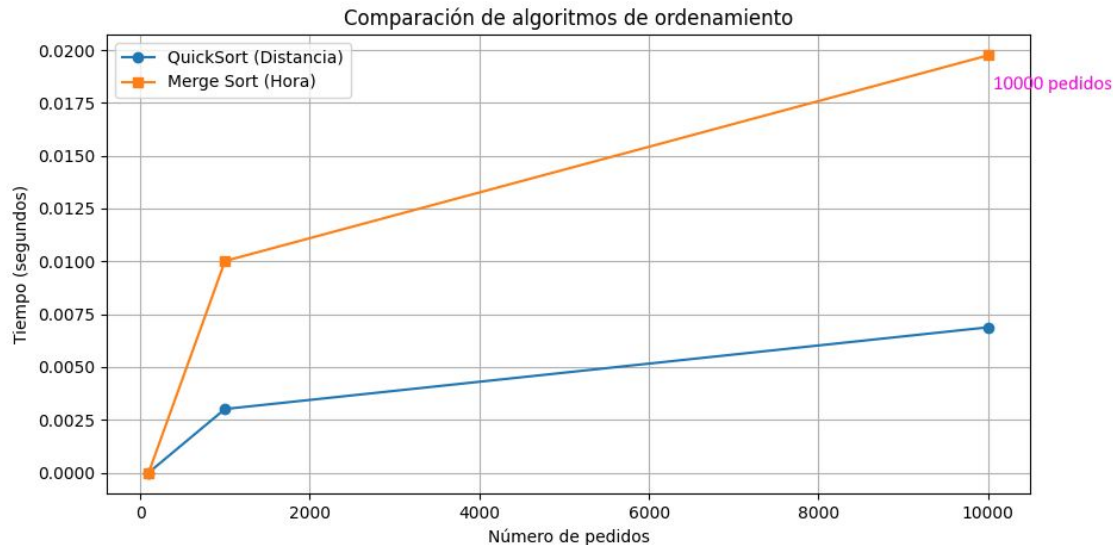
# Resultados Obtenidos

## Gráficos comparativos

- QuickSort: 0.0033 seg
- Merge Sort: 0.099 seg



QuickSort fué 66.6% más veloz



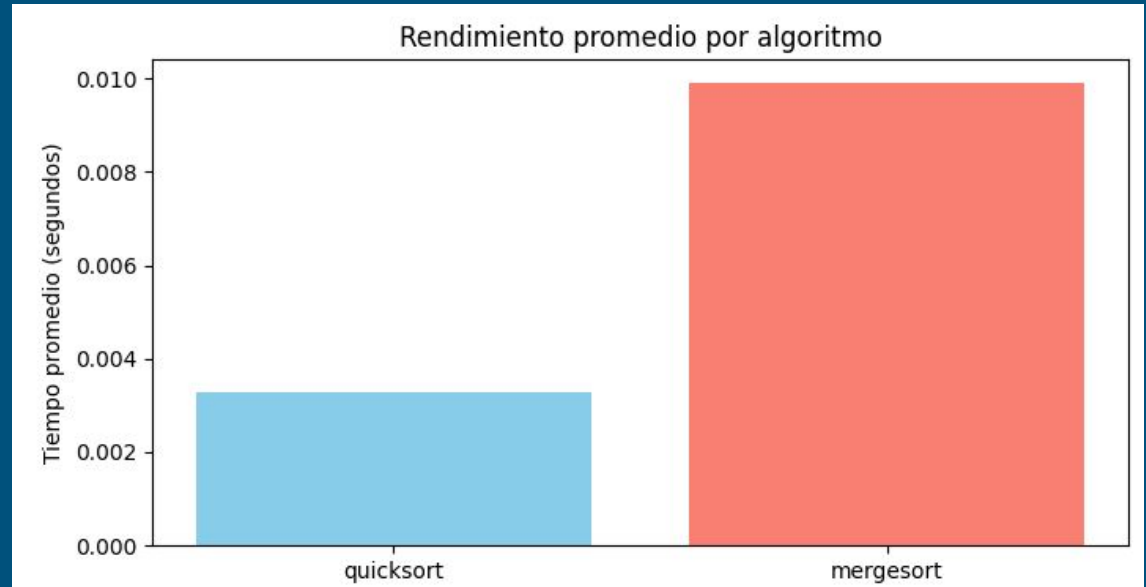
# Resultados Obtenidos

## Gráficos comparativos

- QuickSort: 0.0033 seg
- Merge Sort: 0.0099 seg



QuickSort fué 66.6% más veloz



# Resultados Obtenidos

## Búsqueda Binaria

- Pedido encontrado en 6 comparaciones ( $O(\log n)$ )
- Alta eficacia en datasets ordenados

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Búsqueda binaria realizó 6 comparaciones

=== Resultados ===

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

# Conclusiones

---

# Conclusiones

---

QuickSort, Merge Sort, y Búsqueda Binaria

# Conclusiones

---

## QuickSort vs Merge Sort



QuickSort fue más veloz en datasets grandes

# Conclusiones

---

## QuickSort vs Merge Sort

- ✓ QuickSort fue más veloz en datasets grandes
- ✓ Merge Sort mostró mayor estabilidad y predictibilidad

# Conclusiones

---

## Búsqueda Binaria



Búsqueda binaria funcionó eficientemente en todos los casos (gracias al orden previo)



No ordenar los datos impacta negativamente en los resultados



# Conclusiones

## Finalizando...

- Elegir bien el algoritmo afecta directamente el rendimiento del sistema
- No hay un algoritmo único ideal: hay trade-offs entre tiempo, memoria y estabilidad
- La teoría algorítmica bien aplicada genera mejor experiencia de usuario final
- A futuro: combinar QuickSort + Insertion Sort o aplicar memorización





**Tecnicatura Universitaria en Programación a Distancia**

**Programación 1**

**1° Cuatrimestre**

**2025**