

UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

ALGORITMOS DE ORDENAMIENTO

PROYECTO: BENCHMARKING

 **IVÁN CALDERÓN GARCÍA**
2337556E

 **DIEGO CHÁVEZ FERREIRA**
2337534J

 **LEÓN MAXIMILIANO GUZMÁN**
1926162A

Dra. Violeta Medina Ríos | Sección 503

◎ OBJETIVO & METODOLOGÍA

02

OBJETIVO GENERAL

Diseñar, implementar y evaluar un banco de pruebas (benchmark) para comparar rendimiento y estabilidad de 4 algoritmos de ordenamiento en C.

El objetivo de este proyecto fue diseñar, implementar y evaluar un banco de pruebas para comparar el rendimiento y la estabilidad de cuatro algoritmos de ordenamiento en C, representando diferentes clases de complejidad (cuadrático, $n \log n$, lineal y híbrido).

MÉTODO & MÉTRICAS

- Repeticiones: cada experimento se ejecutó 3 veces y se promediaron los resultados.
- Métricas: **time_ms, comparaciones, movimientos y estable**.
- Tamaños evaluados: 100, 200, 500, 1000, 2500, 5000, 7500.

ALGORITMOS SELECCIONADOS

- **Insertion Sort ($O(n^2)$)**: Eficiente en conjuntos pequeños.
- **Merge Sort ($O(n \log n)$)**: Estable y consistente.
- **Counting Sort ($O(n)$)**: Lineal, sin comparaciones.
- **Introsort (Híbrido)**: Quick + Heap + Insertion.

DATASETS (ENTRADAS)

Tamaños (N): 100, 200, 500, 1000, 2500, 5000, 7500.

Distribuciones:

- Uniforme (Aleatorio)
- Ordenado (1..N)
- Reverso (N..1)
- Casi Ordenado (5% swaps)
- Duplicados (Rango reducido)

Ver generadores: [utilidades.c](#)

HARDWARE DE PRUEBAS

03

EQUIPO I (Principal)

CPU: AMD Ryzen 3 3250U @ 2.595GHz
GPU: Radeon Graphics (4)
RAM: 16GB
OS: Ubuntu 22.04.5 LTS (on Win10)
CC: GCC Compiler

EQUIPO II (Secundario)

CPU: AMD Ryzen 5 3500U
GPU: AMD ATI Radeon Vega Series
RAM: 8GB
OS: Ubuntu 24.04.3 LTS
CC: GCC Compiler

</> ESTRUCTURAS DE DATOS

04

METRICAS.H

Estructura clave pasada por referencia para contar operaciones atómicas sin afectar el flujo del algoritmo.

src/metricas.h

[Ver código completo: metricas_h.html](#)

```
#ifndef ALGORITMOS_H
#define ALGORITMOS_H

typedef struct {
    long long comparaciones;
    long long movimientos;
} Metricas;

void ordenamientoInsercion(int arr[], int n, Metricas* m);
void ordenamientoMezcla(int arr[], int izq, int der, Metricas* m);
void ordenamientoConteo(int arr[], int n, Metricas* m);
void ordenamientoIntro(int arr[], int n, Metricas* m);

#endif
```

⬇️ INSERTION SORT

05

LÓGICA (DETALLE DEL CÓDIGO C)

- **Firma:** void ordenamientoInsercion(int arr[], int n, Metricas* m) — se pasa el arreglo, su tamaño y un contador de métricas por referencia.
- **Bucle externo:** itera *i* desde 1 hasta *n*-1; cada iteración toma la *llave* (*llave* = *arr*[*i*]) y la inserta en la sublistas ordenada a la izquierda.
- **Bucle interno:** con *j* = *i* - 1 se desplazan los elementos mayores que *llave* hacia la derecha (*arr*[*j*+1] = *arr*[*j*]) hasta encontrar la posición correcta.
- **Actualización de métricas:** el código incrementa *m-comparaciones* antes del **while** y dentro del bucle cuando se comprueba la condición; también incrementa *m-movimientos* cada vez que se asigna un valor al arreglo.
- **Estabilidad y espacio:** Insertion Sort es estable (no reordena elementos iguales) y es in-place ($O(1)$ memoria adicional aparte de **Metricas**).
- **Complejidad:** Mejor caso $O(n)$ (ya ordenado), promedio y peor caso $O(n^2)$ (reverso). Ideal para pequeñas *N* o casi ordenados.
- **Riesgos / notas:** el mayor coste son las asignaciones dentro del **while**; el contador de movimientos contabiliza cada asignación explícita en el código.

src/ordenamiento_insercion.c

[Ver código completo: ordenamiento_insercion.c](#)

```
void ordenamientoInsercion(int arr[], int n, Metricas* m) {  
    int i, llave, j;  
    for (i = 1; i < n; i++) {  
        llave = arr[i];  
        m->movimientos++;  
        j = i - 1;  
  
        m->comparaciones++;  
        while (j ≥ 0 && arr[j] > llave) {  
            arr[j + 1] = arr[j];  
            m->movimientos++;  
            j = j - 1;  
            if (j ≥ 0) m->comparaciones++;  
        }  
        arr[j + 1] = llave;  
        m->movimientos++;  
    }  
}
```

💡 MERGE SORT

06

LÓGICA (DETALLE DEL CÓDIGO C)

- **Funciones clave:** rutina recursiva que divide (`ordenamientoMezcla/merge`) y la función `mezclar` que combina dos subarreglos ordenados.
- **Dividir:** la función recibe índices `izq`, `med`, `der` y calcula tamaños `n1`, `n2` para crear arrays temporales `L` y `R`.
- **Mezclar:** usa índices `i`, `j`, `k` y compara `L[i] <= R[j]`; copia el menor al arreglo original y actualiza `m->movimientos` y `m->comparaciones` según corresponda.
- **Estabilidad:** al usar la condición `<=` y copiar desde el subarreglo izquierdo cuando hay empate, el algoritmo mantiene el orden relativo — por eso es estable.
- **Memoria:** requiere $O(n)$ adicional para los arrays temporales (`L` y `R`).
- **Complejidad:** $O(n \log n)$ en todos los casos (mejor/promedio/peor) en tiempo; útil cuando la estabilidad es necesaria.
- **Implementación práctica:** el código incluye copias de los restos tras el bucle principal y actualiza métricas por cada movimiento y comparación relevante.

src/ordenamiento_mezcla.c (Extracto)

[Ver código completo: ordenamiento_mezcla.c](#)

```
void mezclar(int arr[], int izq, int med, int der, Metricas* m) {
    // ... Alloc L y R ...
    i = 0; j = 0; k = izq;
    while (i < n1 && j < n2) {
        m->comparaciones++;
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            m->movimientos++;
            i++;
        } else {
            arr[k] = R[j];
            m->movimientos++;
            j++;
        }
        k++;
    }
    // ... Copiar restantes ...
}
```

↓¹ COUNTING SORT

07

LÓGICA (DETALLE DEL CÓDIGO C)

- **Idea:** crear un array `conteo` de tamaño k (máximo valor esperado + 1) y contar la frecuencia de cada valor presente en `arr`.
- **Fases del algoritmo:**
 1. Contar frecuencias: `for (i=0; i<n; i++) conteo[arr[i]]++;`
 2. Acumular: transformar `conteo` para que cada posición indique la posición final (prefijo acumulado).
 3. Reconstruir salida: iterar el arreglo original en reversa (`for (i=n-1; i>=0; i--)`) y colocar cada elemento en su posición en `salida[--conteo[arr[i]]]` para garantizar estabilidad.
- **Estabilidad:** la reconstrucción en orden inverso preserva estabilidad (los elementos iguales mantienen orden relativo).
- **Complejidad:** $O(n + k)$ tiempo, $O(k)$ memoria adicional; muy eficiente cuando k es comparable a n o menor.
- **Limitaciones:** no es práctico para valores de rango muy grande (memoria elevada). No usa comparaciones directas, por eso las métricas de comparaciones estarán a cero o irrelevantes.

src/ordenamiento_conteo.c

[Ver código completo: ordenamiento_conteo.c](#)

```
void ordenamientoConteo(int arr[], int n, Metricas* m) {  
    // 1. Contar  
    for (i = 0; i < n; i++) conteo[arr[i]]++;  
  
    // 2. Acumular  
    for (i = 1; i <= max; i++) conteo[i] += conteo[i - 1];  
  
    // 3. Construir (Reverso para estabilidad)  
    for (i = n - 1; i >= 0; i--) {  
        salida[conteo[arr[i]] - 1] = arr[i];  
        m->movimientos++;  
        conteo[arr[i]]--;  
    }  
}
```



LÓGICA (DETALLE DEL CÓDIGO C)

- **Estructura general:** la función `introRecursivo(int arr[], int ini, int fin, int prof, Metricas* m)` controla la recursión y decide la estrategia según el tamaño y la profundidad.
- **Caso base (pequeño):** si $n = fin - ini + 1 \leq 16$, se llama a `ordenamientoInsercion` para esa partición (menor overhead para subarreglos pequeños).
- **Control de profundidad:** `prof` inicia como $2 * \log(n)$ desde la llamada principal; si llega a 0, el algoritmo llama a `ordenamientoHeap` para evitar degeneración de Quicksort.
- **Particionado:** en el caso estándar se partitiona (pivot) y se llama recursivamente a las dos mitades con `prof-1` — esto implementa la lógica Quick + fallback a Heap.
- **Complejidad:** promedio $O(n \log n)$ con comportamiento garantizado $O(n \log n)$ por el fallback a Heapsort; no es estable debido a particionados y operaciones de Heap.
- **Ventajas prácticas:** combina la velocidad de Quicksort en la práctica con la seguridad frente a entradas adversas y la eficiencia de Insertion para subproblemas pequeños.

src/ordenamiento_intro.c

[Ver código completo: ordenamiento_intro.c](#)

```
void introRecursivo(int arr[], int ini, int fin, int prof, Metricas* m) {
    int n = fin - ini + 1;
    if (n <= 16) {
        ordenamientoInsercion(arr + ini, n, m);
        return;
    }
    if (prof == 0) {
        ordenamientoHeap(arr + ini, n, m);
        return;
    }
    int piv = particion(arr, ini, fin, m);
    introRecursivo(arr, ini, piv - 1, prof - 1, m);
    introRecursivo(arr, piv + 1, fin, prof - 1, m);
}
```

 **BENCHMARK ENGINE**

09

src/main.c (Bucle Principal)

[Ver código completo: main.c](#)

```
for (algo = 0; algo < 4; algo++) {  
    for (dist = 0; dist < 5; dist++) {  
        for (tam = 0; tam < 7; tam++) {  
            // Generar Datos  
            original = generarDatos(dist, tam);  
            copia = malloc( ... );  
  
            // Medir Tiempo  
            t_inicio = clock();  
            ejecutar(algo, copia, ... );  
            t_fin = clock();  
  
            // Guardar en CSV  
            fprintf(csv, ... );  
        }  
    }  
}
```

Makefile

[Ver Makefile completo](#)

```
all: benchmark  
  
benchmark: src/main.o ...  
           gcc src/main.o ... -o benchmark -lm  
  
src/main.o: src/main.c  
           gcc -Wall -O2 -c src/main.c ...  
  
clean:  
      rm -f src/*.o benchmark
```

RESULTADOS: UNIFORME

10

TIEMPO (MS)

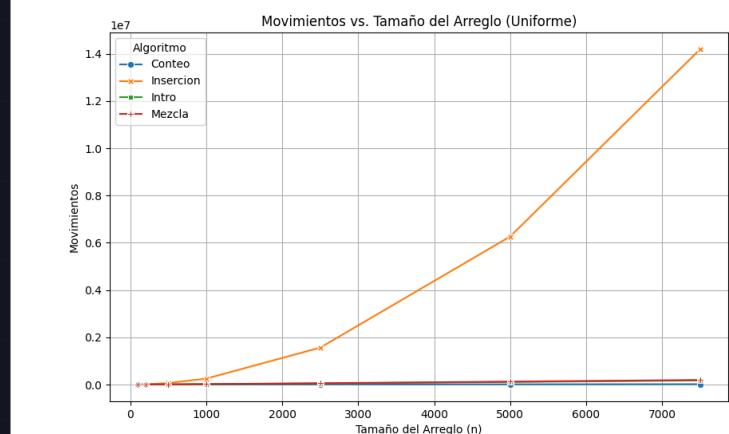
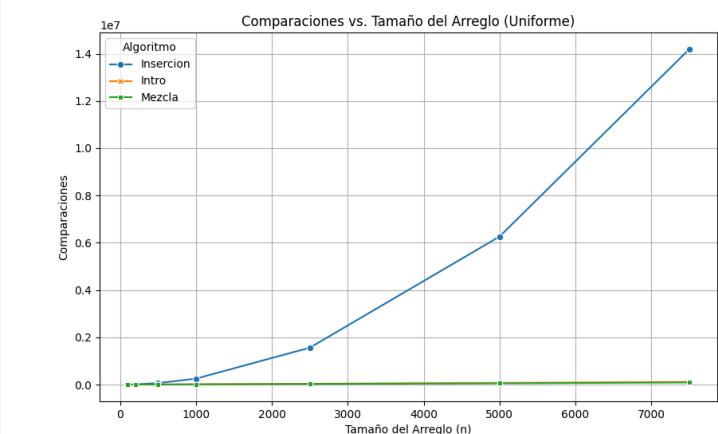
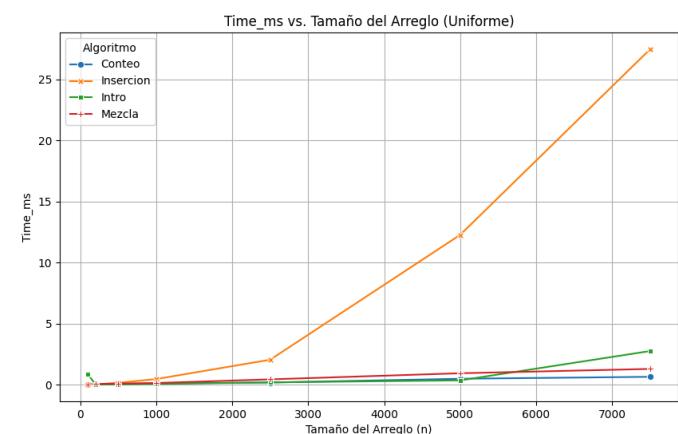
Algoritmo	N=7500
Insertion	22.469
Merge	2.048
Counting	0.631
Intro	0.574

COMPARACIONES

Algoritmo	N=7500
Insertion	14,250,048
Merge	87,099
Intro	110,042

MOVIMIENTOS

Algoritmo	N=7500
Insertion	14,257,555
Merge	193,616
Counting	15,000



◀ RESULTADOS: REVERSO (PEOR CASO)

11

TIEMPO (MS)

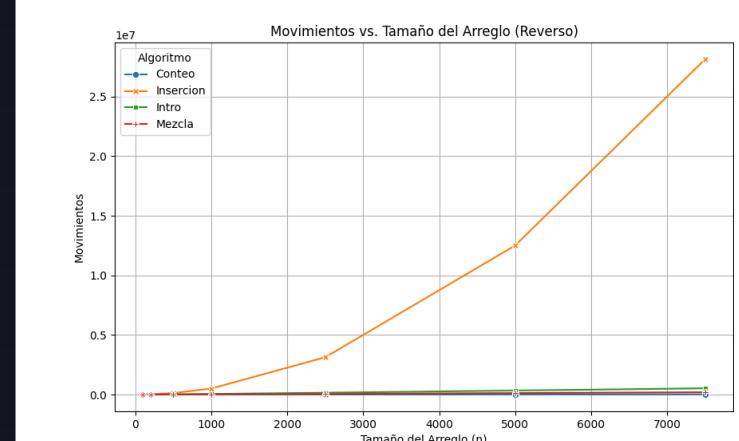
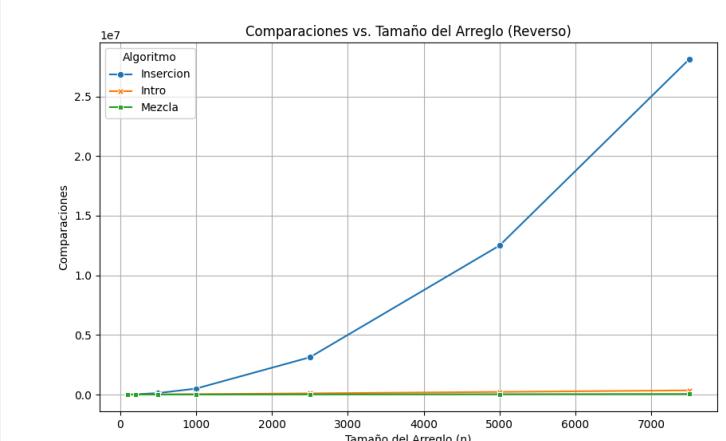
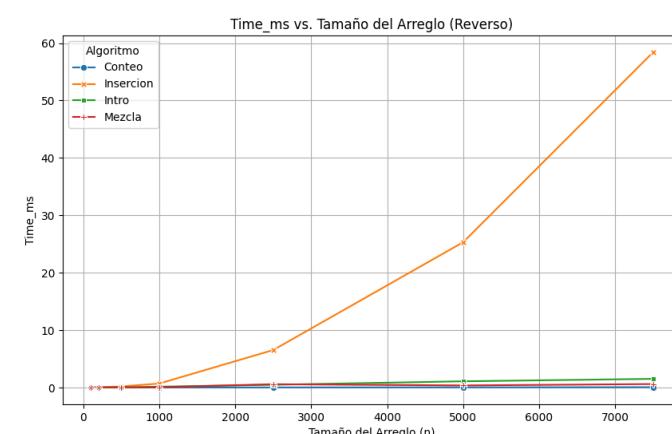
Algoritmo	N=7500
Insertion	64.426
Merge	0.628
Counting	0.063
Intro	5.647

COMPARACIONES

Algoritmo	N=7500
Insertion	28M+
Merge	47,376
Intro	363,049

MOVIMIENTOS

Algoritmo	N=7500
Insertion	28M+
Merge	193,616
Intro	847,680



⬇️ RESULTADOS: CASI ORDENADO

12

TIEMPO (MS)

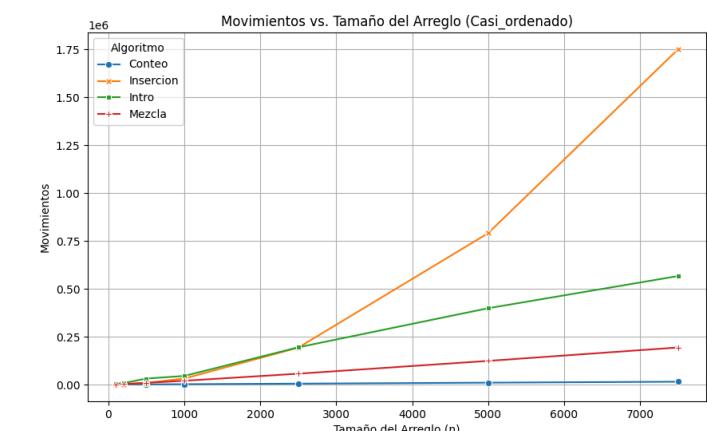
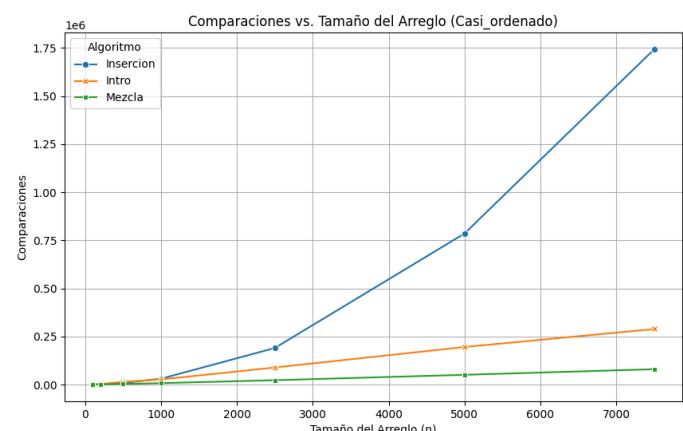
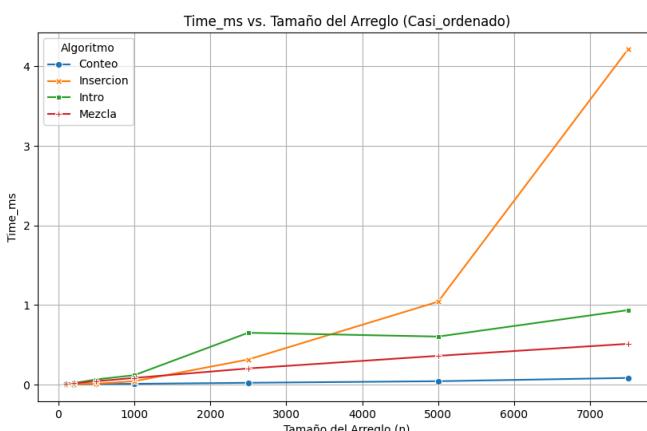
Algoritmo	N=7500
Insertion	2.682
Merge	0.525
Counting	0.105
Intro	0.974

COMPARACIONES

Algoritmo	N=7500
Insertion	1.7M
Merge	79,942
Intro	319,588

ANÁLISIS

Insertion sort brilla aquí. Aunque hace más comparaciones teóricas, su bajo overhead y la naturaleza de los datos lo hacen muy competitivo.



RESULTADOS: DUPLICADOS

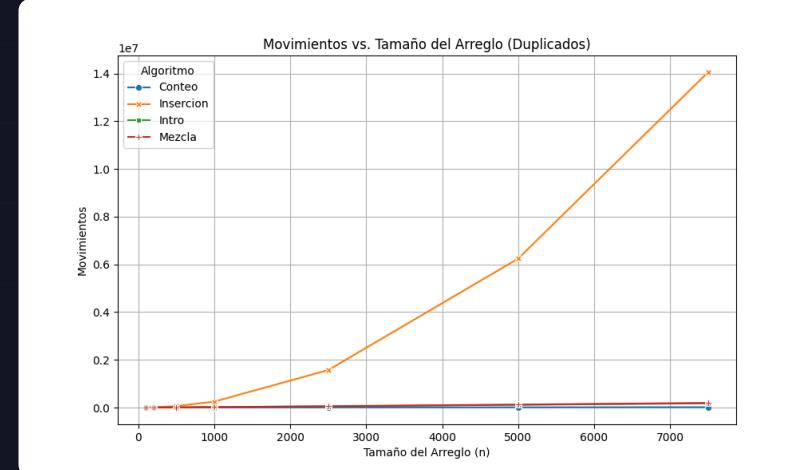
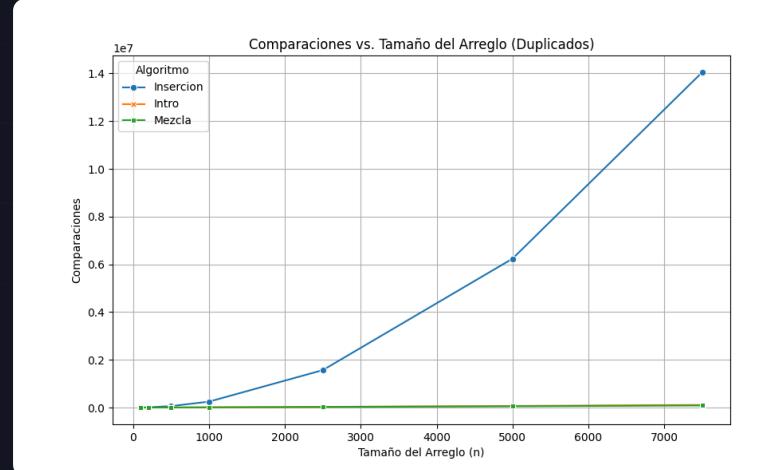
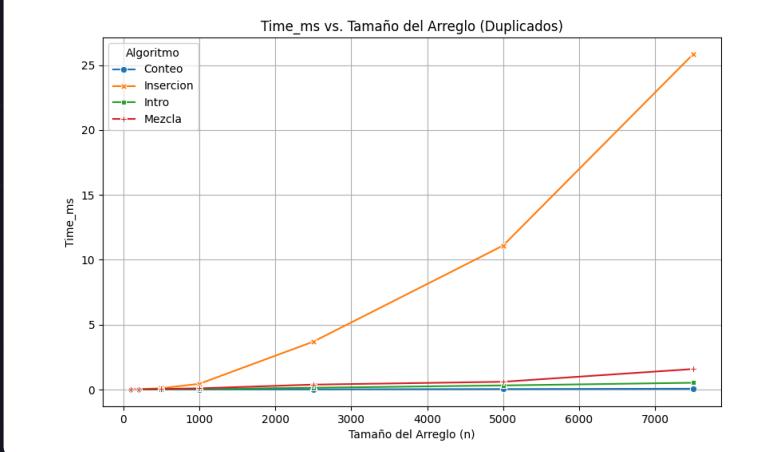
13

TIEMPO (MS) - N=7500

Algoritmo	Tiempo
Insertion	28.028
Merge	1.443
Counting	0.085
Intro	0.502

PRUEBA DE ESTABILIDAD

Algoritmo	¿Estable?
Insertion	SI
Merge	SI
Counting	SI
Introsort	NO



🔍 DISCUSIÓN CRÍTICA

14

$O(n^2)$ VS $O(n \log n)$

En casos "Uniforme" y "Duplicados", las gráficas muestran que Insertion Sort crece exponencialmente comparado con Merge e Introsort.

EL PEOR CASO (REVERSO)

Introsort demostró su robustez. Mientras un Quicksort simple se degradaría a $O(n^2)$, Introsort activó Heapsort manteniendo el rendimiento similar a Merge Sort.

EL ESPECIALISTA: COUNTING SORT

Es el ganador absoluto en tiempo (línea plana en gráficas). No aparece en gráficas de comparaciones porque no las realiza.

EL MEJOR CASO: CASI ORDENADO

¡Sorpresa! Insertion Sort supera expectativas. Al tener pocos elementos fuera de lugar, actúa casi como $O(n)$, superando a algoritmos más complejos debido a su menor sobrecarga (overhead).

❖ CONCLUSIONES & RECOMENDACIONES

16

CONCLUSIONES PRINCIPALES

- Introsort es el algoritmo más robusto en general: combina rapidez y tolerancia al peor caso.
- Counting Sort ofrece los mejores tiempos cuando el rango de valores es pequeño y conocido.
- Insertion Sort es la mejor opción para arreglos casi ordenados o con pocos elementos fuera de lugar.
- Merge Sort garantiza estabilidad y tiempos consistentes en entradas diversas.
- Las métricas recogidas (**time_ms, comparaciones, movimientos**) permiten evaluar trade-offs entre velocidad y operaciones internas.

RECOMENDACIONES PRÁCTICAS

- Usar **Introsort** como algoritmo por defecto en bibliotecas generales debido a su balance entre velocidad y seguridad frente al peor caso.
- Aplicar **Counting Sort** para datasets de enteros con rango limitado (cuando la memoria adicional es aceptable).
- Emplear **Insertion** como subrutina en híbridos para particiones pequeñas o como algoritmo rápido en arreglos casi ordenados.
- Elegir **Merge Sort** cuando la estabilidad sea un requisito funcional (p. ej. orden por múltiples claves).
- Para benchmarking reproducible: fijar semillas, ejecutar múltiples repeticiones y reportar medianas/medias con desviación.

SCRIPT DE VISUALIZACIÓN

16

scripts/graficar.py

[Ver script completo: graficar.py](#)

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv("resultados.csv")
metricas = ['time_ms', 'comparaciones', 'movimientos']

for dist in df['distribucion'].unique():
    subset = df[df['distribucion'] == dist]

    for metrica in metricas:
        plt.figure(figsize=(10, 6))
        sns.lineplot(data=subset, x='tamano', y=metrica, hue='algoritmo')
        plt.title(f"{metrica} vs Tamaño ({dist})")
        plt.savefig(f"graficas/{dist}_{metrica}.png")
```

GRACIAS POR SU ATENCION

¿PREGUNTAS?

```
Process finished with exit code 0
```