

# ALGORITMOS DE ORDENAMIENTO

## PROYECTO: BENCHMARKING



IVÁN CALDERÓN GARCÍA  
2337556E



DIEGO CHÁVEZ FERREIRA  
2337534J



LEÓN MAXIMILIANO GUZMÁN  
1926162A

DESARROLLADO POR:

## OBJETIVO & METODOLOGÍA

### 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)

## IPSO I (Principal)

PU: AMD Ryzen 3 3250U @ 2.595GHz

PU: Radeon Graphics (4)

RAM: 16GB

S: Ubuntu 22.04.5 LTS (on Win10)  
C: GCC Compiler

IPO II (Secundario)

## ESTRUCTURAS DE DATOS

METRICAS.H

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

/metricas.h  
• código completo: [metricas\\_h.html](#)

```
fndef ALGORITMOS_H
efine ALGORITMOS_H

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

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

ndif
```

## INSERTION SORT

LOGICA (DETALLE DEL CODIGO 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.

/ordenamiento\_insercion.c

```
> código completo: ordenamiento_insercion.c
```

```
id 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 ÓGICA (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.

```
/ordenamiento_mezcla.c (Extracto)
```

```
> código completo: ordenamiento_mezcla.c
```

```
id 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 ...
```

## COUNTING SORT ÓGICA (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.

**Límites:** 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.

```
/ordenamiento_conteo.c
```

```
> código completo: ordenamiento_conteo.c
```

```
id ordenamientoConceo(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]]--;
}

```

## INTROSORT 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  $2^{\lfloor \log_2(n) \rfloor}$  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.

```
/ordenamiento_intro.c
```

```
* código completo: ordenamiento_intro.c
```

```

int 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);
}

```

```
/main.c (Bucle Principal)
```

```
* código completo: main.c
```

```

r (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, ...);
    }
}

```

```
efile
```

## IMPORTE TANOS: UNIFORME

Porcentaje	N=7500	
Porcentaje		22.469
Porcentaje		2.048
Porcentaje		0.631
Porcentaje		0.574

## IMPARACIONES

Porcentaje	N=7500	
Porcentaje		14,250,048
Porcentaje		87,099
Porcentaje		110,042

## MVIMIENTOS

Porcentaje	N=7500	
Porcentaje		14,257,555
Porcentaje		193,616
Porcentaje		15,000

## IMPORTE TANOS: REVERSO (PFOR CASO)

Porcentaje	N=7500	
Porcentaje		64.426
Porcentaje		0.628
Porcentaje		0.063
Porcentaje		5.647

## IMPARACIONES

Porcentaje	N=7500	
Porcentaje		28M+
Porcentaje		47,376
Porcentaje		363,049

## MVIMIENTOS

Porcentaje	N=7500	
Porcentaje		28M+
Porcentaje		193,616
Porcentaje		847,680

## IMPORTE TANOS: CASI ORDENADO

Porcentaje	N=7500	
Porcentaje		2.682
Porcentaje		0.525
Porcentaje		0.105
Porcentaje		0.974

## COMPARACIONES

Algoritmo	N=7500
Conteo	1.7M
Introducción	79,942
Merge	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.

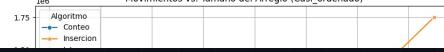
Time\_ms vs. Tamaño del Arreglo (Casi\_ordenado)



Comparaciones vs. Tamaño del Arreglo (Casi\_ordenado)



Movimientos vs. Tamaño del Arreglo (Casi\_ordenado)



## TIEMPOS EN TIEMPOS DE TIPIFICACIÓN

Algoritmo	Tiempo
Conteo	28.028
Introducción	1.443
Merge	0.085
Quicksort	0.502

## PRUEBA DE ESTABILIDAD

Algoritmo	¿Estable?
Conteo	SI
Introducción	SI
Merge	SI
Quicksort	NO

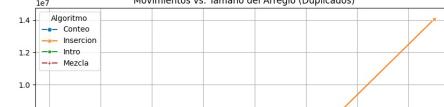
Time\_ms vs. Tamaño del Arreglo (Duplicados)



Comparaciones vs. Tamaño del Arreglo (Duplicados)



Movimientos vs. Tamaño del Arreglo (Duplicados)



## DISCUSIÓN CRÍTICA

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

### 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

```
ipts/graficar.py
? script completo: graficar.py

port pandas as pd
port seaborn as sns
port matplotlib.pyplot as plt

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

r 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
```