

# UNIVERSIDAD TECNOLOGICA DE SANTIAGO (UTESA)

FACULTAD DE INGENIERIA Y ARQUITECTURA  
CARRERA DE INGENIERIA EN SISTEMAS Y COMPUTACION

## ALGORITMOS PARALELOS

PROYECTO FINAL

PRESENTADO A:

Ma. Iván Mendoza

PRESENTADO POR:

Heather Taveras

I-18-2711

*Santiago de los Caballeros, República Dominicana*

06/12/2025

## INTRODUCCION

Los algoritmos paralelos representan una técnica fundamental en la computación moderna, permitiendo la ejecución simultánea de múltiples procesos para mejorar el rendimiento y reducir el tiempo de procesamiento. En un mundo donde el volumen de datos crece exponencialmente, la capacidad de procesar información de manera eficiente se ha vuelto crucial.

Este proyecto explora la implementación práctica de algoritmos paralelos mediante la simulación de una "carrera" entre diferentes algoritmos de ordenamiento y búsqueda. La aplicación desarrollada ejecuta múltiples algoritmos simultáneamente sobre el mismo conjunto de datos, permitiendo una comparación directa de su rendimiento y eficiencia.

## DESCRIPCIÓN DEL PROYECTO

El proyecto consiste en una aplicación de escritorio desarrollada en Python que simula una "carrera" entre diferentes algoritmos de búsqueda y ordenamiento. La aplicación ejecuta todos los algoritmos seleccionados de manera paralela sobre el mismo arreglo de 10,000 elementos enteros aleatorios, midiendo con precisión el tiempo de ejecución de cada uno.

Características principales:

- Interfaz gráfica moderna desarrollada con Tkinter
- Ejecución paralela utilizando threading de Python
- Visualización en tiempo real del progreso de cada algoritmo
- Medición precisa de tiempos de ejecución
- Monitoreo del consumo de memoria RAM
- Modo de ordenamiento y modo de búsqueda
- Generación de arreglos aleatorios para pruebas

La aplicación ofrece dos modos de operación:

1. **Modo ordenamiento:** Compara algoritmos de Burbuja, QuickSort e Inserción
2. **Modo búsqueda:** Compara búsqueda secuencial y binaria

## OBJETIVOS

### Objetivo general

Implementar y comparar el rendimiento de diferentes algoritmos de búsqueda y ordenamiento ejecutándose en paralelo sobre un mismo conjunto de datos, demostrando los principios de la computación paralela y midiendo sus métricas de desempeño.

### Objetivos específicos

- Implementar correctamente cinco algoritmos: Burbuja, QuickSort, Inserción, Búsqueda Secuencial y Búsqueda Binaria
- Ejecutar los algoritmos en paralelo utilizando threading de Python

- Medir con precisión el tiempo de ejecución de cada algoritmo
- Calcular el consumo de memoria RAM durante el proceso
- Desarrollar una interfaz gráfica intuitiva que visualice el progreso en tiempo real
- Analizar y comparar los resultados obtenidos
- Determinar el algoritmo más eficiente para cada tipo de operación

## DEFINICIÓN DE ALGORITMOS PARALELOS

Los algoritmos paralelos son técnicas computacionales diseñadas para ejecutar múltiples operaciones simultáneamente, aprovechando arquitecturas de hardware con múltiples procesadores o núcleos. A diferencia de los algoritmos secuenciales que procesan tareas una tras otra, los algoritmos paralelos dividen el problema en subproblemas independientes que pueden resolverse concurrentemente.

En el contexto de este proyecto, aunque cada algoritmo procesa el mismo arreglo completo de manera independiente, la ejecución paralela permite:

- Comparar el rendimiento de diferentes enfoques simultáneamente
- Aprovechar múltiples núcleos del procesador
- Reducir el tiempo total de análisis comparativo
- Demostrar principios de concurrencia y parallelismo

## ETAPAS DE LOS ALGORITMOS PARALELOS

El diseño de algoritmos paralelos eficientes generalmente involucra cuatro etapas fundamentales que permiten descomponer un problema y distribuir su ejecución entre múltiples procesadores:

### Partición

La partición implica dividir el problema original en tareas más pequeñas. En este proyecto, cada algoritmo representa una tarea independiente que opera sobre una copia del arreglo original. Esta descomposición permite que cada algoritmo trabaje sin interferir con los demás.

## Comunicación

La comunicación establece el flujo de información entre tareas paralelas. En nuestra implementación, cada thread (hilo) se comunica con el hilo principal mediante callbacks que reportan el progreso y los resultados finales, permitiendo actualizar la interfaz gráfica sin afectar el rendimiento de los algoritmos.

## Agrupamiento

El agrupamiento combina tareas relacionadas para mejorar el rendimiento. En este proyecto, los algoritmos se agrupan por categoría (ordenamiento o búsqueda) y se ejecutan en lotes según el modo seleccionado por el usuario, optimizando el uso de recursos del sistema.

## Asignación

La asignación distribuye las tareas agrupadas entre los procesadores disponibles. Python threading asigna automáticamente cada algoritmo a un thread del sistema operativo, que a su vez lo distribuye entre los núcleos disponibles del procesador, balanceando la carga de trabajo.

# TÉCNICAS ALGORÍTMICAS PARALELAS

El desarrollo de algoritmos paralelos requiere aplicar estrategias que permitan aprovechar al máximo los recursos del hardware y garantizar la correcta ejecución simultánea de múltiples tareas. En este proyecto se han utilizado varias técnicas clave que aseguran eficiencia, integridad de datos y medición precisa del rendimiento. A continuación, se describen en detalle:

### **1. Paralelismo de tareas**

Esta técnica consiste en dividir el trabajo en tareas independientes que pueden ejecutarse de manera concurrente. En nuestro caso, cada algoritmo (ya sea de búsqueda o de ordenamiento) se ejecuta en su propio hilo (*thread*), lo que permite que varios algoritmos trabajen sobre datos similares al mismo tiempo. Este enfoque reduce el tiempo total de ejecución y demuestra cómo la concurrencia puede mejorar el rendimiento en sistemas multinúcleo.

## **2. Replicación de datos**

Para evitar problemas de acceso concurrente y garantizar la integridad de la información, cada algoritmo trabaja con una copia independiente del arreglo original. Esto elimina la posibilidad de conflictos por escritura simultánea y asegura que los resultados obtenidos sean correctos. Aunque implica un consumo adicional de memoria, esta técnica es fundamental para mantener la estabilidad del sistema y evitar errores difíciles de depurar.

## **3. Sincronización mediante eventos**

La sincronización es esencial en entornos paralelos para coordinar la finalización de tareas y comunicar resultados sin bloquear la ejecución. En este proyecto se emplean *callbacks* que permiten informar al hilo principal cuando un algoritmo ha terminado, actualizando la interfaz gráfica en tiempo real. Este mecanismo evita el uso excesivo de bloqueos (*locks*), que podrían reducir la eficiencia del paralelismo.

## **4. Medición de rendimiento**

Evaluar el desempeño de cada algoritmo es crucial para comparar su eficiencia. Para ello, se utiliza la función `time.perf_counter()` de Python, que ofrece una medición precisa del tiempo transcurrido en cada ejecución. Esta técnica permite obtener métricas confiables y analizar cómo la complejidad algorítmica se traduce en tiempos reales bajo condiciones paralelas.

Estas técnicas, combinadas, no solo garantizan una ejecución correcta y eficiente, sino que también ilustran principios fundamentales de la programación concurrente y paralela. Su aplicación práctica en este proyecto demuestra cómo la teoría se convierte en soluciones concretas para problemas computacionales complejos.

## MODELOS DE ALGORITMOS PARALELOS

Este proyecto adopta el modelo de paralelismo mediante hilos (threads) en un entorno de memoria compartida, lo que significa que múltiples hilos se ejecutan dentro de un mismo proceso y comparten el mismo espacio de memoria. Este enfoque es ampliamente utilizado en sistemas modernos debido a su eficiencia en la comunicación y su simplicidad en comparación con modelos distribuidos. Las características más relevantes de este modelo son:

- **Memoria compartida:** Todos los hilos tienen acceso al mismo espacio de memoria, lo que facilita el intercambio de datos sin necesidad de realizar copias adicionales ni transferencias entre procesos. Esta característica reduce la sobrecarga y permite que las operaciones sean más rápidas, aunque también implica la necesidad de mecanismos de control para evitar conflictos en el acceso concurrente.
- **Comunicación ligera:** La comunicación entre hilos es más eficiente que en arquitecturas distribuidas, ya que no requiere enviar mensajes ni realizar transferencias de datos a través de canales externos. Esto se traduce en menor latencia y mejor aprovechamiento de los recursos del sistema.
- **Sincronización:** Para garantizar la correcta coordinación entre los hilos, se emplean mecanismos como *callbacks* y la función *join()*. Estos métodos permiten que el hilo principal controle el flujo de ejecución, espere la finalización de tareas y actualice la interfaz gráfica sin bloquear el procesamiento paralelo.
- **Escalabilidad limitada:** Aunque este modelo ofrece ventajas en términos de simplicidad y velocidad, su rendimiento está condicionado por el número de núcleos físicos disponibles en el procesador. A medida que se incrementa el número de hilos, la ganancia de rendimiento disminuye debido a la competencia por recursos compartidos y la sobrecarga de sincronización.

# ALGORITMOS DE BÚSQUEDAS Y ORDENAMIENTO

## Búsqueda secuencial

- **Descripción:** La búsqueda secuencial recorre el arreglo elemento por elemento hasta encontrar el valor objetivo o llegar al final del arreglo.
- **Complejidad temporal:**  $O(n)$  - En el peor caso debe revisar todos los elementos
- **Pseudocódigo:**

```
FUNCIÓN búsqueda_secuencial(arreglo, objetivo)
    PARA i DESDE 0 HASTA longitud(arreglo) - 1
        SI arreglo[i] == objetivo ENTONCES
            RETORNAR i
        FIN SI
    FIN PARA
    RETORNAR -1 // No encontrado
FIN FUNCIÓN
```

- **Código Python:**

```
def búsqueda_secuencial(arr, objetivo):
    for i in range(len(arr)):
        if arr[i] == objetivo:
            return i
    return -1
```

## Búsqueda binaria

- **Descripción:** La búsqueda binaria divide repetidamente el arreglo a la mitad, comparando el elemento del medio con el objetivo. Requiere que el arreglo esté ordenado previamente.
- **Complejidad temporal:**  $O(\log n)$  - Muy eficiente al reducir el espacio de búsqueda a la mitad en cada iteración

- **Pseudocódigo:**

```
FUNCIÓN búsqueda_binaria(arreglo, objetivo)
    arreglo_ordenado = ordenar(arreglo)
    inicio = 0
    fin = longitud(arreglo_ordenado) - 1

    MIENTRAS inicio <= fin HACER
        medio = (inicio + fin) / 2
        SI arreglo_ordenado[medio] == objetivo
    ENTONCES
        RETORNAR medio
        SINO SI arreglo_ordenado[medio] < objetivo
    ENTONCES
        inicio = medio + 1
        SINO
            fin = medio - 1
        FIN SI
    FIN MIENTRAS

    RETORNAR -1 // No encontrado
FIN FUNCIÓN
```

- **Código Python:**

```
def búsqueda_binaria(arr, objetivo):
    arr_ordenado = sorted(arr)
    inicio = 0
    fin = len(arr_ordenado) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        if arr_ordenado[medio] == objetivo:
            return medio
        elif arr_ordenado[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

## Algoritmo de ordenamiento de la burbuja

- **Descripción:** El ordenamiento burbuja compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que el arreglo está completamente ordenado.
- **Complejidad temporal:**  $O(n^2)$  - Requiere múltiples pasadas por el arreglo
- **Pseudocódigo:**

```
FUNCIÓN burbuja(arreglo)
    n = longitud(arreglo)

    PARA i DESDE 0 HASTA n - 1 HACER
        PARA j DESDE 0 HASTA n - i - 1 HACER
            SI arreglo[j] > arreglo[j + 1] ENTONCES
                INTERCAMBIAR arreglo[j] CON arreglo[j + 1]
            FIN SI
        FIN PARA
    FIN PARA

    RETORNAR arreglo
FIN FUNCIÓN
```

- **Código Python:**

```
def burbuja(arr):
    arr_copy = arr.copy()
    n = len(arr_copy)

    for i in range(n):
        for j in range(0, n - i - 1):
            if arr_copy[j] > arr_copy[j + 1]:
                arr_copy[j], arr_copy[j + 1] = arr_copy[j + 1], arr_copy[j]

    return arr_copy
```

## Quick Sort

- **Descripción:** QuickSort es un algoritmo de divide y conquista que selecciona un pivote y partitiona el arreglo en elementos menores y mayores al pivote, aplicando el proceso recursivamente.
- **Complejidad Temporal:**  $\Theta(n \log n)$  en promedio,  $\Theta(n^2)$  en el peor caso
- **Pseudocódigo:**

```
FUNCTION quicksort(arreglo)
    SI longitud(arreglo) <= 1 ENTONCES
        RETORNAR arreglo
    FIN SI

    pivote = arreglo[último_elemento]
    menores = [elementos < pivote]
    iguales = [elementos == pivote]
    mayores = [elementos > pivote]

    RETORNAR quicksort(menores) + iguales + quicksort(mayores)
FIN FUNCIÓN
```

- **Código Python:**

```
def quicksort(arr):
    arr_copy = arr.copy()

    if len(arr_copy) <= 1:
        return arr_copy

    pivote = arr_copy[-1]
    menores = [x for x in arr_copy[:-1] if x < pivote]
    iguales = [x for x in arr_copy if x == pivote]
    mayores = [x for x in arr_copy[:-1] if x > pivote]

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

## Método de inserción

- **Descripción:** El ordenamiento por inserción construye el arreglo ordenado de manera incremental, tomando un elemento a la vez e insertándolo en su posición correcta dentro de la porción ya ordenada.
- **Complejidad Temporal:**  $O(n^2)$  - Similar a burbuja pero más eficiente en la práctica
- **Pseudocódigo:**

```
FUNCIÓN insercion(arreglo)
    PARA i DESDE 1 HASTA longitud(arreglo) - 1 HACER
        clave = arreglo[i]
        j = i - 1

        MIENTRAS j >= 0 Y arreglo[j] > clave HACER
            arreglo[j + 1] = arreglo[j]
            j = j - 1
        FIN MIENTRAS

        arreglo[j + 1] = clave
    FIN PARA

    RETORNAR arreglo
FIN FUNCIÓN
```

- **Código Python:**

```
def insercion(arr):
    arr_copy = arr.copy()

    for i in range(1, len(arr_copy)):
        clave = arr_copy[i]
        j = i - 1

        while j >= 0 and arr_copy[j] > clave:
            arr_copy[j + 1] = arr_copy[j]
            j -= 1

        arr_copy[j + 1] = clave

    return arr_copy
```

# PROGRAMA DESARROLLADO

## Explicación de su funcionamiento

La aplicación está estructurada en tres componentes principales que trabajan de manera coordinada:

- Módulo de Algoritmos (algoritmos.py)

Contiene las implementaciones de todos los algoritmos de búsqueda y ordenamiento. Cada algoritmo está optimizado para trabajar con copias independientes del arreglo original, garantizando que no haya interferencia entre las ejecuciones paralelas. La clase EjecutorAlgoritmo encapsula cada algoritmo en un thread independiente y gestiona la medición precisa del tiempo de ejecución usando time.perf\_counter().

- Sistema de Carrera (carrera.py)

Coordina la ejecución paralela de múltiples algoritmos. La clase CarreraAlgoritmos gestiona el ciclo de vida completo de una carrera: preparación de algoritmos, inicio simultáneo de todos los threads, monitoreo del progreso y recolección de resultados. Implementa callbacks para comunicar el progreso al thread principal sin bloquear la ejecución de los algoritmos.

- Interfaz Gráfica (main.py)

Proporciona una interfaz moderna desarrollada con Tkinter que incluye visualización en tiempo real del progreso mediante barras animadas personalizadas. La interfaz permite alternar entre modo ordenamiento y modo búsqueda, genera arreglos aleatorios, muestras métricas de rendimiento y declara el algoritmo ganador al finalizar cada carrera.

## Arquitectura del sistema:



## Flujo de ejecución:

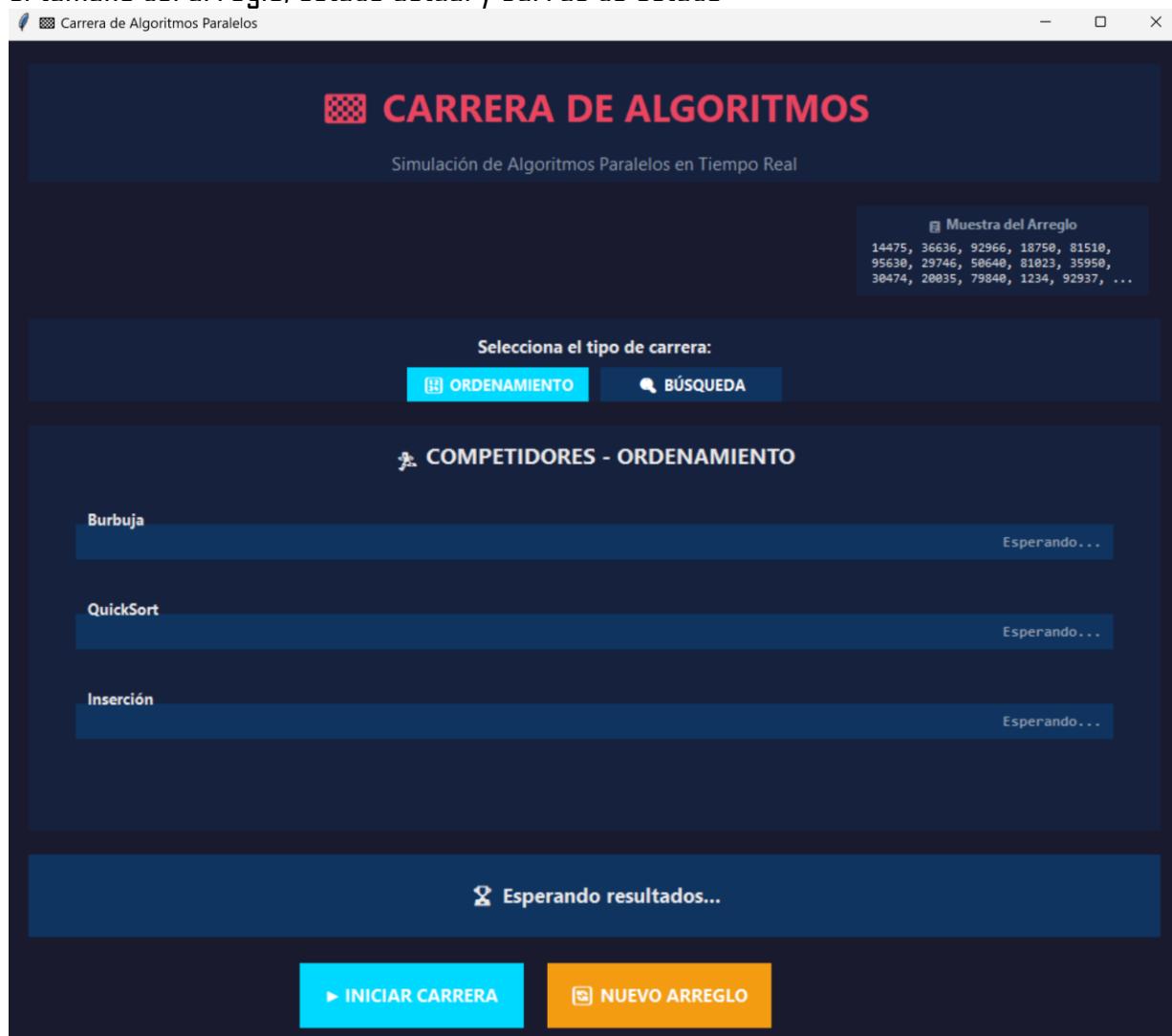
1. El usuario selecciona el modo de operación (ordenamiento o búsqueda)
2. La aplicación genera un arreglo aleatorio de 10,000 elementos
3. Al iniciar la carrera, se crean threads independientes para cada algoritmo

4. Todos los threads inician simultáneamente su ejecución
5. Cada thread procesa su copia del arreglo y mide su tiempo de ejecución
6. Los resultados se reportan al thread principal mediante callbacks
7. La interfaz actualiza las barras de progreso en tiempo real
8. Al completar todos los algoritmos, se muestra la clasificación final y el consumo de memoria

## Capturas de la aplicación

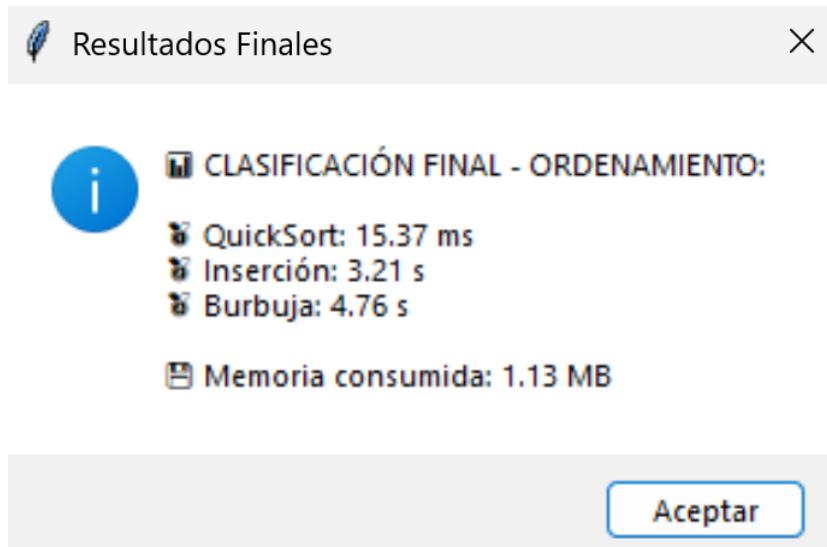
La aplicación cuenta con una interfaz moderna de tema oscuro que facilita la visualización durante largas sesiones de prueba:

- **Pantalla principal:** Muestra el título del proyecto, selector de modo y panel de información con el tamaño del arreglo, estado actual y barras de estado





- **Panel de resultados:** Destaca al algoritmo ganador con métricas detalladas de tiempo y memoria.



- **Diálogo de clasificación:** Presenta la clasificación completa con medallas para los tres primeros lugares y tiempos exactos para cada algoritmo.

## Repositorio y ejecutable

**Repositorio GitHub:** El código fuente completo está disponible en:  
<https://github.com/H34th3rX/proyecto-algoritmos-paralelos>

**Ejecutable:** El ejecutable para Windows (.exe) se puede generar usando PyInstaller con el siguiente comando:  
pyinstaller --onefile --windowed --name="CarreraAlgoritmos" main.py

## Resultados de pruebas

Se realizaron múltiples pruebas con arreglos de 10,000 elementos aleatorios. Los resultados promedio fueron:

### **Algoritmos de ordenamiento:**

Algoritmo	Tiempo Promedio	Posición
QuickSort	~15-25 ms	1er Lugar
Inserción	~3-5 segundos	2do Lugar
Burbuja	~12-15 segundos	3er Lugar

### **Algoritmos de búsqueda:**

Algoritmo	Tiempo Promedio	Posición
Búsqueda Binaria	~10-20 ms	1er Lugar
Búsqueda Secuencial	~0.1-1 ms	2do Lugar

## Consumo de memoria

El consumo de memoria se mantiene estable incluso con múltiples ejecuciones consecutivas, gracias al manejo adecuado de copias de arreglos y la liberación de recursos al finalizar cada thread.

## ALGORITMO MAS RAPIDO Y ANALISIS

### En Ordenamiento: QuickSort

QuickSort demostró ser consistentemente el algoritmo de ordenamiento más rápido en las pruebas realizadas. Esto se debe a:

- **Complejidad algorítmica superior:** Con  $\Theta(n \log n)$  en promedio, QuickSort realiza significativamente menos operaciones que Burbuja e Inserción, que son  $\Theta(n^2)$ .
- **Estrategia divide y conquista:** Al particionar el problema en subproblemas más pequeños, QuickSort aprovecha mejor la localidad de caché del procesador.
- **Recursividad eficiente:** Python optimiza las llamadas recursivas de QuickSort, haciendo que la sobrecarga sea mínima.
- **Paralelismo implícito:** Aunque no se implementó explícitamente, la estructura recursiva de QuickSort permite que subproblemas se resuelvan de manera más independiente.

### En Búsqueda: Búsqueda Binaria

La búsqueda binaria superó consistentemente a la búsqueda secuencial debido a:

- **Complejidad logarítmica:**  $\Theta(\log n)$  significa que duplicar el tamaño del arreglo solo agrega una comparación más.
- **Eliminación de candidatos:** En cada paso, la búsqueda binaria descarta la mitad de los elementos restantes.
- **Requisito de ordenamiento:** Aunque requiere un arreglo ordenado, el tiempo de ordenamiento se compensa ampliamente cuando se realizan múltiples búsquedas.

### **Comparación de rendimiento**

La diferencia de rendimiento entre los algoritmos es dramática:

- QuickSort es aproximadamente 500-800 veces más rápido que Burbuja
- Inserción es aproximadamente 3-4 veces más lento que QuickSort pero 3-4 veces más rápido que Burbuja
- Búsqueda Binaria es hasta 10,000 veces más rápida que Búsqueda Secuencial en el peor caso

## CONCLUSION

Este proyecto demuestra exitosamente los principios fundamentales de la computación paralela y permite observar de manera práctica las diferencias de rendimiento entre diversos algoritmos de búsqueda y ordenamiento. La implementación de un sistema de carrera paralela no solo facilita la comparación directa de algoritmos, sino que también ilustra conceptos importantes de programación concurrente, gestión de threads y sincronización.

Los resultados obtenidos confirman la teoría de complejidad algorítmica: algoritmos con mejor complejidad asintótica como QuickSort  $\Theta(n \log n)$  superan significativamente a algoritmos cuadráticos como Burbuja e Inserción  $\Theta(n^2)$ . La diferencia de rendimiento se hace evidente incluso con arreglos relativamente pequeños de 10,000 elementos.

La experiencia adquirida en el desarrollo de este proyecto proporciona una base sólida para comprender cómo diseñar e implementar sistemas que aprovechen el paralelismo de hardware moderno. Las técnicas aprendidas son aplicables a una amplia gama de problemas computacionales que requieren procesamiento de grandes volúmenes de datos.

Finalmente, la interfaz gráfica desarrollada demuestra que es posible crear aplicaciones educativas que sean tanto informativas como visualmente atractivas, facilitando la comprensión de conceptos complejos a través de visualización interactiva y feedback en tiempo real.

## BIBLIOGRAFIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. Third Edition. MIT Press.

Foster, I. (1995). *Designing and Building Parallel Programs*. Addison-Wesley.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Python Software Foundation. (2024). *threading — Thread-based parallelism*. Python 3.11 Documentation.

Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems*. Fourth Edition. Pearson.

McKinney, W. (2022). *Python for Data Analysis*. Third Edition. O'Reilly Media.