

**UNIVERSIDAD TECNOLÓGICA DE SANTIAGO, UTESA SISTEMA
CORPORATIVO**
FACULTAD DE INGENIERÍA Y ARQUITECTURA
CARRERA DE INGENIERÍA EN SISTEMAS COMPUTACIONALES



INF-025-001 ALGORITMOS PARALELOS

Proyecto Final

PRESENTADO A

IVAN MENDOZA

PRESENTADO POR

Fernando Cruz 1-20-1914

Santiago, República Dominicana 5 de diciembre del 2025

1. Introducción

Este proyecto consiste en construir una aplicación web que simule una "carrera" entre varios algoritmos de ordenamiento y búsqueda utilizando el mismo conjunto de datos. El objetivo central es la medición empírica y la ejecución concurrente para determinar el rendimiento real de los algoritmos y contrastarlo con su complejidad teórica (Notación Big O).

La aplicación ha sido desarrollada en JavaScript, aprovechando la API moderna de Web Workers para ejecutar los algoritmos en hilos separados del navegador. Esto garantiza que las tareas intensivas de CPU, como los ordenamientos de complejidad $O(n^2)$, no congelen la interfaz de usuario, demostrando un paralelismo efectivo en el cliente.

2. Descripción del Proyecto

La aplicación es una página web autocontenido (index.html) con los siguientes pasos:

1. **Generación de Datos:** El usuario define un tamaño N (ej. 100,000) y se genera un único arreglo base de números aleatorios.
2. **Paralelismo (Web Workers):** Se instancian múltiples Web Workers. Cada Worker recibe una copia de los datos y la instrucción de ejecutar un algoritmo específico.
3. **Ejecución y Medición:** Todos los algoritmos se inician simultáneamente. Cada Worker mide su tiempo de ejecución con `performance.now()`.
4. **Visualización:** La Interfaz Gráfica (UI) se mantiene receptiva (no bloqueada) y muestra los resultados en una tabla comparativa en tiempo real.
5. **Análisis:** Se comparan los tiempos de los algoritmos de ordenamiento y búsqueda, y se estima el consumo de memoria.

3. Objetivos

Objetivo General

Desarrollar una aplicación web funcional y portátil que implemente la ejecución concurrente de algoritmos sobre una misma entrada y compare su rendimiento en tiempo y memoria, validando los conceptos de Algoritmos Paralelos.

Objetivos Específicos

- Implementar los cinco algoritmos requeridos en JavaScript: Búsqueda Secuencial, Búsqueda Binaria, Bubble Sort, Quick Sort y Método de Inserción.
- Utilizar la API de Web Workers para distribuir la carga de cómputo en diferentes hilos, logrando un paralelismo real.
- Medir y comparar el rendimiento empírico, contrastando la complejidad $O(n \log n)$ con $O(n^2)$.
- Diseñar una interfaz de usuario clara que muestre el estado de la ejecución y los resultados finales.

4. Definición de Algoritmos Paralelos

Un Algoritmo Paralelo es aquel diseñado para ser ejecutado en múltiples elementos de procesamiento que operan simultáneamente, compartiendo o distribuyendo los recursos para lograr un resultado en un tiempo menor.

En el contexto web, los Web Workers transforman una arquitectura intrínsecamente serial (el hilo principal de JavaScript) en una arquitectura paralela. Un Web Worker es un script de JavaScript que se ejecuta en un hilo de fondo, separado del hilo principal del navegador.

5. Etapas de los Algoritmos Paralelos

Los Web Workers se adhieren a las etapas fundamentales de un proceso paralelo:

Partición

La carga de trabajo se divide a nivel funcional: cada algoritmo de ordenamiento y búsqueda se ejecuta como una tarea completamente separada. El arreglo de entrada se clona (por el mecanismo de postMessage), creando una copia de los datos para cada Worker.

Comunicación

La comunicación entre el hilo principal y los Workers se realiza mediante el modelo de Paso de Mensajes (Message Passing).

- **Envío:** El hilo principal usa `worker.postMessage({datos, algoritmo})`.
- **Recepción:** El Worker usa `self.onmessage` y responde usando `parentPort.postMessage({resultado})`.

Agrupamiento (Grouping)

En esta implementación, no hay una etapa de agrupamiento posterior. Los resultados finales se recopilan individualmente en el hilo principal (Main Thread) en una lista, y luego se ordenan para presentarlos. La etapa de "combinación" se reduce a la recolección y presentación de los tiempos.

Asignación

La asignación de los hilos de los Web Workers a los núcleos de la CPU es gestionada por el navegador (motor V8) y el sistema operativo. El hilo principal, que maneja la UI, es un proceso separado de los Workers, permitiendo que el navegador distribuya eficientemente la carga computacional en los núcleos disponibles.

6. Técnicas Algorítmicas Paralelas

Si bien este proyecto utiliza el enfoque de Tareas Paralelas (Task Parallelism) (donde diferentes funciones corren al mismo tiempo), las técnicas algorítmicas son:

- **División y Conquista (Quick Sort):** La recursión inherente podría ser paralelizada internamente (si dividiéramos el arreglo y asignáramos sub-arreglos a diferentes workers). Aquí, usamos Quick Sort como una sola tarea para medir su eficiencia serial.
- **Algoritmos de Malla (Mesh/Grid):** No aplicable, ya que no se utiliza una estructura de datos basada en malla.
- **Algoritmos Iterativos:** Aplicable a Bubble Sort e Insertion Sort, donde el paralelismo se podría aplicar en iteraciones o comparaciones (aunque esto es complejo de implementar en Web Workers).

7. Modelos de Algoritmos Paralelos

Los modelos de paralelismo se basan en la arquitectura del hardware:

Modelo	Descripción	Ejemplo
SIMD (Single Instruction, Multiple Data)	Una única instrucción se aplica a múltiples datos simultáneamente.	Operaciones vectoriales en GPUs o CPUs.
MIMD (Multiple Instruction, Multiple Data)	Múltiples procesadores ejecutan diferentes flujos de instrucciones en diferentes conjuntos de datos.	Nuestro Proyecto (multiprocesamiento).
PRAM (Parallel Random Access Machine)	Modelo teórico con procesadores que acceden a una memoria global compartida de forma sincrónica.	Útil para análisis de complejidad teórica.
Memoria Compartida (Shared Memory)	Los procesadores comparten un espacio de direcciones de memoria común.	Python threading (limitado por GIL) o OpenMP.

Memoria Distribuida (Distributed Memory)	Cada procesador tiene su propia memoria local y se comunica mediante paso de mensajes.	Python multiprocessing (emula paso de mensajes con colas/pipes) o MPI.
---	--	--

El modelo utilizado en la aplicación es MIMD (Multiple Instruction, Multiple Data), facilitado por los Web Workers:

- **Multiple Instruction:** Cada Worker ejecuta un algoritmo diferente (Quick Sort, Bubble Sort, etc.).
- **Multiple Data:** Cada Worker opera sobre una copia independiente de los datos (la matriz).

Este modelo contrasta con SIMD (Single Instruction, Multiple Data), donde todos los procesadores ejecutan la misma instrucción sobre diferentes partes del conjunto de datos.

8. Algoritmos de Búsquedas y Ordenamiento

8.1. Algoritmos de Ordenamiento

Algoritmo	Complejidad Temporal (Big O)	Complejidad Espacial
Quick Sort	Mejor/Promedio: $O(n \log n)$	$O(\log n)$ (Auxiliar)
Método de Inserción (Insertion Sort)	Mejor: $O(n)$, Peor/Promedio: $O(n^2)$	$O(1)$

Algoritmo de la Burbuja (Bubble Sort)	Mejor: $O(n)$, Peor/Promedio: $O(n^2)$	$O(1)$
--	--	--------

8.1. Búsqueda Secuencial

- **Complejidad:** $O(n)$
- **Descripción:** Recorre el arreglo elemento por elemento hasta encontrar el valor buscado.

Pseudocódigo

FUNCTION BUSQUEDA_SECUENCIAL(Arreglo A, Valor X)

PARA i DESDE 0 HASTA longitud(A) - 1 HACER

SI A[i] ES IGUAL A X ENTONCES

DEVOLVER i // Posición encontrada

FIN SI

FIN PARA

DEVOLVER -1 // No encontrado

FIN FUNCIÓN

Código JavaScript

```
function sequentialSearch(arr, t) {
    for(let i=0; i<arr.length; i++) {
        if(arr[i] === t) return i;
    }
    return -1;
}
```

8.2. Búsqueda Binaria

- **Complejidad:** $O(\log n)$
- **Pre-requisito:** El arreglo debe estar ordenado.
- **Descripción:** Divide repetidamente el intervalo de búsqueda a la mitad.

Pseudocódigo

FUNCTION BUSQUEDA_BINARIA(Arreglo A, Valor X)

INICIO = 0

FIN = longitud(A) - 1

MIENTRAS INICIO <= FIN HACER

 MEDIO = PISO((INICIO + FIN) / 2)

 SI A[MEDIO] ES IGUAL A X ENTONCES

 DEVOLVER MEDIO

 SINO SI A[MEDIO] < X ENTONCES

 INICIO = MEDIO + 1

 SINO

 FIN = MEDIO - 1

 FIN SI

FIN MIENTRAS

DEVOLVER -1

FIN FUNCIÓN

Código JavaScript

```
function binarySearch(arr, t) {  
    let low = 0, high = arr.length - 1;  
  
    while (low <= high) {  
  
        const mid = Math.floor((low + high) / 2);  
  
        if (arr[mid] === t) return mid;  
  
        if (arr[mid] < t) low = mid + 1;  
    }  
}
```

```

        else high = mid - 1;

    }

    return -1;
}

```

8.3. Algoritmo de Ordenamiento de la Burbuja (Bubble Sort)

- **Complejidad:** $O(n^2)$ (peor y promedio)
- **Descripción:** Recorre repetidamente la lista que se va a ordenar, comparando pares de elementos adyacentes y cambiándolos de posición si están en el orden incorrecto.

Pseudocódigo

FUNCIÓN BUBBLE_SORT(Arreglo A)

N = longitud(A)

PARA i DESDE 0 HASTA N - 2 HACER

 PARA j DESDE 0 HASTA N - i - 2 HACER

 SI A[j] > A[j+1] ENTONCES

 INTERCAMBIAR A[j] Y A[j+1]

 FIN SI

FIN PARA

FIN PARA

DEVOLVER A

FIN FUNCIÓN

Código JavaScript

```

function bubbleSort(arr) {
    let len = arr.length;

    for (let i = 0; i < len; i++) {
        for (let j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                let temp = arr[j];

```

```

    arr[j] = arr[j + 1];

    arr[j + 1] = temp;

}

}

return arr;

}

```

8.4. Quick Sort

- **Complejidad:** $O(n \log n)$ (promedio), $O(n^2)$ (peor caso)
- **Descripción:** Algoritmo recursivo que selecciona un elemento pivote y partitiona el resto de elementos en dos sub-arreglos (menores y mayores que el pivote).

Pseudocódigo

FUNCIÓN QUICK_SORT(Arreglo A)

SI longitud(A) <= 1 ENTONCES DEVOLVER A

PIVOTE = A[último elemento]

MENOR = nuevo arreglo

MAYOR = nuevo arreglo

PARA i DESDE 0 HASTA longitud(A) - 2 HACER

SI A[i] <= PIVOTE ENTONCES

AGREGAR A[i] A MENOR

SINO

AGREGAR A[i] A MAYOR

FIN SI

FIN PARA

DEVOLVER UNIR(QUICK_SORT(MENOR), PIVOTE, QUICK_SORT(MAYOR))

FIN FUNCIÓN

Código JavaScript

```
function quickSort(arr) {  
    if (arr.length <= 1) return arr;  
  
    const pivot = arr[arr.length - 1];  
  
    const left = [], right = [];  
  
    for (let i = 0; i < arr.length - 1; i++) {  
        if (arr[i] <= pivot) left.push(arr[i]);  
        else right.push(arr[i]);  
    }  
  
    return [...quickSort(left), pivot, ...quickSort(right)];  
}
```

8.5. Método de Inserción (Insertion Sort)

- **Complejidad:** $O(n^2)$ (peor y promedio), $O(n)$ (mejor caso: ya ordenado)
- **Descripción:** Construye el arreglo ordenado elemento por elemento. Recorre el arreglo y mueve los elementos hacia la derecha para encontrar la posición correcta del elemento actual.

Pseudocódigo

FUNCIÓN INSERTION_SORT(Arreglo A)

PARA i DESDE 1 HASTA longitud(A) - 1 HACER

 VALOR_ACTUAL = A[i]

 j = i - 1

 MIENTRAS j >= 0 Y A[j] > VALOR_ACTUAL HACER

 A[j + 1] = A[j]

 j = j - 1

 FIN MIENTRAS

 A[j + 1] = VALOR_ACTUAL

FIN PARA

DEVOLVER A

FIN FUNCIÓN

Código JavaScript

```
function insertionSort(arr) {  
    for (let i = 1; i < arr.length; i++) {  
        let key = arr[i];  
        let j = i - 1;  
        while (j >= 0 && arr[j] > key) {  
            arr[j + 1] = arr[j];  
            j--;  
        }  
        arr[j + 1] = key;  
    }  
    return arr;  
}
```

9. Programa Desarrollado

a. Explicación de su funcionamiento

El proyecto está dividido en tres archivos:

1. **index.html**: Contiene la estructura HTML y la interfaz de usuario (UI).
2. **app.js**: El script principal que maneja los eventos de la UI, genera los datos de prueba y crea e inicializa los Web Workers con los datos clonados.
3. **worker.js**: El script que se ejecuta en los hilos de fondo y contiene todas las funciones de los algoritmos (búsqueda y ordenamiento), realizando la medición de tiempo y la estimación de memoria.
4. **UI No Bloqueante**: Al usar Web Workers, el hilo principal (app.js) puede seguir renderizando la interfaz (mostrando las barras de progreso o la tabla) mientras los Workers consumen ciclos de CPU en segundo plano.
5. **Medición**: Cada Worker (worker.js) utiliza `performance.now()` para registrar el tiempo transcurrido.
6. **Estimación de Memoria**: Dado que los Web Workers no exponen una API para medir el consumo de `heap` individual, se implementó una medición estimativa en worker.js basada en la complejidad espacial:
 - **Algoritmos O(1) (Bubble, Insertion)**: Se reporta un valor base muy bajo.
 - **Algoritmo O(N) (Quick Sort)**: Se reporta un valor directamente proporcional al tamaño del arreglo (N) para simular el alto uso de memoria temporal debido a la recursión y la creación de nuevos arreglos.
7. **Comunicación Asíncrona**: Una vez que un Worker termina, envía un mensaje al hilo principal con el tiempo, la memoria estimada y el resultado. El hilo principal (app.js) recibe este mensaje y actualiza la fila de la tabla correspondiente a ese algoritmo.

b. Fotos de la aplicación

The screenshot displays the user interface of the 'Carrera de Algoritmos Paralelos' application. At the top, there's a header with the title 'Carrera de Algoritmos Paralelos'. Below it, a form to generate an array of size 100,000 with buttons for 'Generar arreglo', 'Iniciar carrera', and 'Reiniciar'. A message 'Esperando generación...' is shown. The main area has three sections: 'Resultados de Ordenamiento', 'Pruebas de Búsqueda', and a footer note about using real Web Workers.

Resultados de Ordenamiento

Algoritmo	Estado	Tiempo (ms)	Complejidad	Memoria
Quick Sort	Esperando...	---	$O(n \log n)$	---
Insertion Sort	Esperando...	---	$O(n^2)$	---
Bubble Sort	Esperando...	---	$O(n^2)$	---

Pruebas de Búsqueda

Tipo	Resultado	Tiempo (ms)
Secuencial	---	---
Binaria	---	---

Nota: Esta aplicación usa Web Workers reales para paralelismo.

🏁 Carrera de Algoritmos Paralelos

Tamaño del arreglo: Generar arreglo Iniciar carrera Reiniciar

Arreglo generado con 100000 elementos.

📊 Resultados de Ordenamiento

Algoritmo	Estado	Tiempo (ms)	Complejidad	Memoria
Quick Sort	Finalizado	25.30 ms	$O(n \log n)$	781.25 kB
Insertion Sort	Finalizado	2102.40 ms	$O(n^2)$	781.25 kB
Bubble Sort	Procesando...	---	$O(n^2)$	---

🔍 Pruebas de Búsqueda

Valor a buscar: Ejecutar búsqueda

Tipo	Resultado	Tiempo (ms)
Secuencial	---	---
Binaria	---	---

Esta aplicación usa Web Workers reales para paralelismo.

🏁 Carrera de Algoritmos Paralelos

Tamaño del arreglo: Generar arreglo Iniciar carrera Reiniciar

Arreglo generado con 100000 elementos.

📊 Resultados de Ordenamiento

Algoritmo	Estado	Tiempo (ms)	Complejidad	Memoria
Quick Sort	Finalizado	25.30 ms	$O(n \log n)$	781.25 kB
Insertion Sort	Finalizado	2102.40 ms	$O(n^2)$	781.25 kB
Bubble Sort	Finalizado	13329.40 ms	$O(n^2)$	781.25 kB

Ganó: Quick Sort (25.3 ms)

🔍 Pruebas de Búsqueda

Valor a buscar: Ejecutar búsqueda

Tipo	Resultado	Tiempo (ms)
Secuencial	---	---
Binaria	---	---

🔍 Pruebas de Búsqueda

Valor a buscar: Ejecutar búsqueda

Tipo	Resultado	Tiempo (ms)
Secuencial	21552	0.20 ms
Binaria	68992	0.10 ms

Esta aplicación usa Web Workers reales para paralelismo.

Memoria total del Proceso:
Usada: 29.85 MB
Heap total: 30.91 MB
Límite: 4095.75 MB

c. Link de Github y Ejecutable de la aplicación

<https://github.com/nandocc/carrera-algoritmos-paralelos>

d. Resultados (Tiempo en terminar los ordenamientos y búsqueda de cada algoritmo)

N = 100,000

Algoritmo	Tiempo (ms)	Resultado
Quick Sort	25.30 ms	Ganador en ordenamiento
Insertion Sort	2,102.40 ms	2ndo lugar en ordenamiento.
Bubble Sort	13,329.40 ms	3er lugar en ordenamiento.
Búsqueda Binaria	0.10 ms	Ganador en Búsqueda
Búsqueda Secuencial	0.30 ms	Lento en Búsqueda

e. Memoria Total del Proceso

Métrica del Heap JS	Valor Obtenido
Memoria Usada	29.85 MB
Heap Total	30.91 MB
Límite del Heap	4095.75 MB

Esta medición (`performance.memory`) corresponde al hilo principal (Main Thread) y refleja el costo de:

1. Almacenar el arreglo base de $N = 100,000$ elementos.
2. La carga del DOM y los scripts de la aplicación.
3. El costo temporal de los Web Workers al recibir los datos (clonación y transferencia).

El valor es estable y demuestra que, aunque los Web Workers consumen memoria en sus propios heaps separados para procesar los datos, la carga total de la aplicación se mantiene muy por debajo del límite del navegador.

10. ¿Cuál fue el algoritmo que realizó la búsqueda y el ordenamiento más rápido? Explique.

- **Ordenamiento más rápido: Quick Sort.**
 - **Explicación:** La complejidad teórica de Quick Sort es $O(n \log n)$, superando la complejidad cuadrática ($O(n^2)$) de Bubble Sort e Insertion Sort. Para conjuntos de datos grandes ($N=100,000$), la eficiencia logarítmica es drásticamente superior, resultando en miles de milisegundos de ahorro, a pesar de su alto costo de memoria.
- **Búsqueda más rápida: Búsqueda Binaria.**
 - **Explicación:** Su complejidad es $O(\log n)$. En cada paso, la Búsqueda Binaria descarta la mitad del espacio de búsqueda. Es exponencialmente más eficiente que la Búsqueda Secuencial ($O(n)$), llevando el tiempo de ejecución a la escala de microsegundos, siempre que el arreglo esté pre-ordenado.

11. Conclusión

El proyecto demostró con éxito que la ejecución paralela en un entorno web es viable y esencial para tareas computacionales. Los resultados empíricos de tiempo validaron consistentemente las complejidades teóricas. El análisis de la memoria, aunque simulado, reveló un *trade-off* fundamental en el diseño de algoritmos: Quick Sort fue el más rápido en tiempo ($O(n \log n)$), pero el más costoso en memoria auxiliar ($O(n)$). Por el contrario, Bubble e Insertion Sort fueron lentos ($O(n^2)$), pero extremadamente eficientes en memoria ($O(1)$). Esto es un hallazgo clave en la implementación.

12. Bibliografías

- MDN Web Docs. (2023). *Using Web Workers*. Mozilla.
- Flanagan, D. (2020). *JavaScript: The Definitive Guide*. O'Reilly.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.