Universidad Tecnológica de Santiago

UTESA.

Sistema corporativo.

Algoritmos Paralelos

INF-025-001

Tema:	Proyecto final
Presentado por:	Rut S. Santos 2-17-1270
Entregado a:	Ing. Iván Mendoza
Fecha de entrega:	23 de Marzo del 2025

Santiago de los Caballeros, República Dominicana.

Tabla de contenido

Introducción

En la actualidad, el rendimiento de las aplicaciones informáticas puede optimizarse mediante algoritmos eficientes y el uso adecuado de la computación paralela o concurrente. Los algoritmos de búsqueda y ordenamiento se utilizan constantemente en múltiples ámbitos (bases de datos, análisis de datos, sistemas de recomendación, etc.).

Este proyecto busca ilustrar cómo distintos algoritmos —de ordenamiento y de búsqueda— pueden competir en una "carrera", midiendo cuál resuelve primero la tarea asignada. Además, se revisa brevemente el marco teórico de los algoritmos paralelos y cómo se relacionan con la división de procesos en sub-tareas.

Descripción del Proyecto

Se ha desarrollado una aplicación que:

- Genera un arreglo de números aleatorios.
- Ejecuta, de manera concurrente, varios algoritmos de ordenamiento (Burbuja, Quick Sort e Inserción) y de búsqueda (Secuencial y Binaria).
- Mide el tiempo de ejecución de cada algoritmo hasta su finalización.
- Muestra los resultados en pantalla (tiempo consumido y cuál algoritmo terminó primero).

El objetivo es comparar la eficiencia de estos algoritmos y también aplicar conceptos de concurrencia (hilos o threads).

Objetivos

1. Objetivo General

Analizar el comportamiento y eficiencia de distintos algoritmos de ordenamiento y búsqueda al ejecutarlos de forma concurrente, determinando cuál presenta el mejor desempeño en tiempo de ejecución.

2. **Objetivos Específicos**
    - Implementar diferentes algoritmos de ordenamiento: Burbuja, Quick Sort e Inserción.
    - Implementar algoritmos de búsqueda: Secuencial y Binaria.
    - Realizar mediciones de tiempo de ejecución para cada algoritmo.
    - Comparar los resultados de cada implementación.
    - Demostrar la factibilidad de ejecutar los algoritmos de manera concurrente (simulando un escenario de algoritmos paralelos).

Definición de Algoritmos Paralelos

Un algoritmo paralelo es aquel que se puede descomponer en subprocesos independientes que pueden ejecutarse simultáneamente en uno o varios procesadores o núcleos de un sistema informático. De esta forma, se busca acelerar la solución de problemas complejos y aprovechar la capacidad de cómputo de las arquitecturas multiprocesador.

En esencia, la paralelización consiste en dividir una tarea en partes que puedan realizarse al mismo tiempo, reduciendo el tiempo global de cómputo frente a una ejecución estrictamente secuencial.

Etapas de los Algoritmos paralelos

1. **Partición**

    Se divide el problema original en subproblemas o tareas que puedan ser trabajadas en paralelo.

2. **Comunicación**

    Implica el intercambio de datos entre los subprocesos/tareas cuando sea necesario. En algunos casos, la comunicación se reduce para mejorar el rendimiento.

3. **Agrupamiento**

    Se reagrupan (o combinan) los resultados parciales provenientes de las distintas tareas. En un ordenamiento paralelo, por ejemplo, podría ser la fusión de subarreglos ordenados.

4. **Asignación**

   Distribución de los subprocesos entre los distintos procesadores o hilos de ejecución disponibles para balancear la carga de trabajo.

Técnicas Algorítmicas Paralelas

Existen varias técnicas para diseñar algoritmos paralelos, por ejemplo:

- Dividir y conquistar (Divide and Conquer): La tarea se fracciona en subproblemas de menor tamaño, que luego se combinan.
- Descomposición funcional: Cada hilo/proceso ejecuta una función distinta en la secuencia de pasos del algoritmo.
- Descomposición de datos: Los datos se dividen entre múltiples procesadores, cada uno realiza la misma tarea sobre su porción de datos.

Modelos de Algoritmos Paralelos

Entre los modelos para diseñar algoritmos paralelos se pueden destacar:

- Modelo PRAM (Parallel Random Access Machine): Un modelo teórico donde varios procesadores comparten memoria global con tiempos de acceso uniforme.
- Modelo de paso de mensajes (MPI): En sistemas distribuidos, se comunican a través de mensajes.
- Modelo de hilos (Threading): Uso de hilos ligeros que comparten memoria en un mismo proceso.

Algoritmos de Búsquedas y Ordenamiento (Adjuntar Pseudocódigo y código de cada uno)

1. **Búsqueda Secuencial**

   a. Pseduocódigo

```
función busqueda_secuencial(arreglo, valor_buscado):
    para i en 0 hasta tamaño(arreglo)-1:
        si arreglo[i] == valor_buscado:
            retornar i
    retornar -1
```

   b. Código Python.

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

2. **Búsqueda Binaria**

   a. Pseduocódigo

```
función busqueda_binaria(arreglo_ordenado, valor_buscado):
    low = 0
    high = longitud(arreglo_ordenado) - 1

    mientras low <= high:
        mid = (low + high) // 2
        si arreglo_ordenado[mid] == valor_buscado:
            retornar mid
        sino si arreglo_ordenado[mid] < valor_buscado:
            low = mid + 1
        de lo contrario:
            high = mid - 1
    retornar -1
```

b. Código python

```
def busqueda_binaria(arr, valor):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == valor:
            return mid
        elif arr[mid] < valor:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

3. Algoritmo de Ordenamiento de la Burbuja

a. Pseudocódigo

```
función bubble_sort(arreglo):
    n = longitud(arreglo)
    para i en 0 hasta n-1:
        para j en 0 hasta n-i-2:
            si arreglo[j] > arreglo[j+1]:
                intercambiar arreglo[j] y arreglo[j+1]
```

b. Código python

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

4. Quick Sort

a. Pseudocódigo

```
función quick_sort(arreglo, inicio, fin):
    si inicio < fin:
        pivote = particion(arreglo, inicio, fin)
        quick_sort(arreglo, inicio, pivote - 1)
        quick_sort(arreglo, pivote + 1, fin)

función particion(arreglo, inicio, fin):
    pivot = arreglo[fin]
    i = inicio - 1
    para j en inicio hasta fin - 1:
        si arreglo[j] <= pivot:
            i = i + 1
            intercambiar arreglo[i] y arreglo[j]
    intercambiar arreglo[i+1] y arreglo[fin]
    retornar i + 1
```

b. Código python

```
def quick_sort(arr):
    def _quick_sort(sub_arr, low, high):
        if low < high:
            pivot_index = partition(sub_arr, low, high)
            _quick_sort(sub_arr, low, pivot_index - 1)
            _quick_sort(sub_arr, pivot_index + 1, high)

    def partition(sub_arr, low, high):
        pivot = sub_arr[high]
    i = low - 1
        for j in range(low, high):
            if sub_arr[j] <= pivot:
                i += 1
                sub_arr[i], sub_arr[j] = sub_arr[j],
sub_arr[i]
        sub_arr[i+1], sub_arr[high] = sub_arr[high],
sub_arr[i+1]
        return i + 1
    _quick_sort(arr, 0, len(arr) - 1)
```

5. **Método de Inserción**

   a. Pseudocódigo

```
función insertion_sort(arreglo):
    para i en 1 hasta longitud(arreglo)-1:
        valor = arreglo[i]
        j = i - 1
        mientras j >= 0 y arreglo[j] > valor:
            arreglo[j+1] = arreglo[j]
            j = j - 1
        arreglo[j+1] = valor
```

   b. Código python

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Programa desarrollado

1. **Explicación de su funcionamiento**

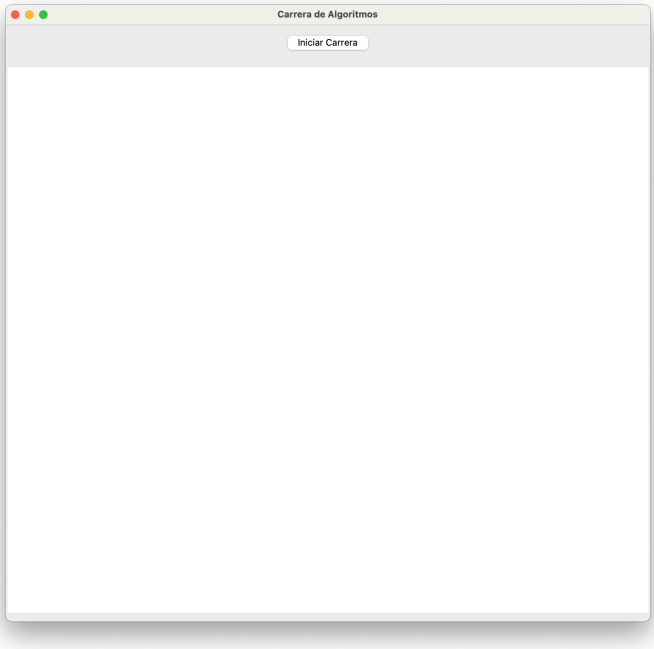
   El programa fue desarrollado en Python, utilizando la librería `tkinter` para la interfaz gráfica y `threading` para ejecutar en paralelo (o, más precisamente, de forma concurrente) cada uno de los algoritmos:

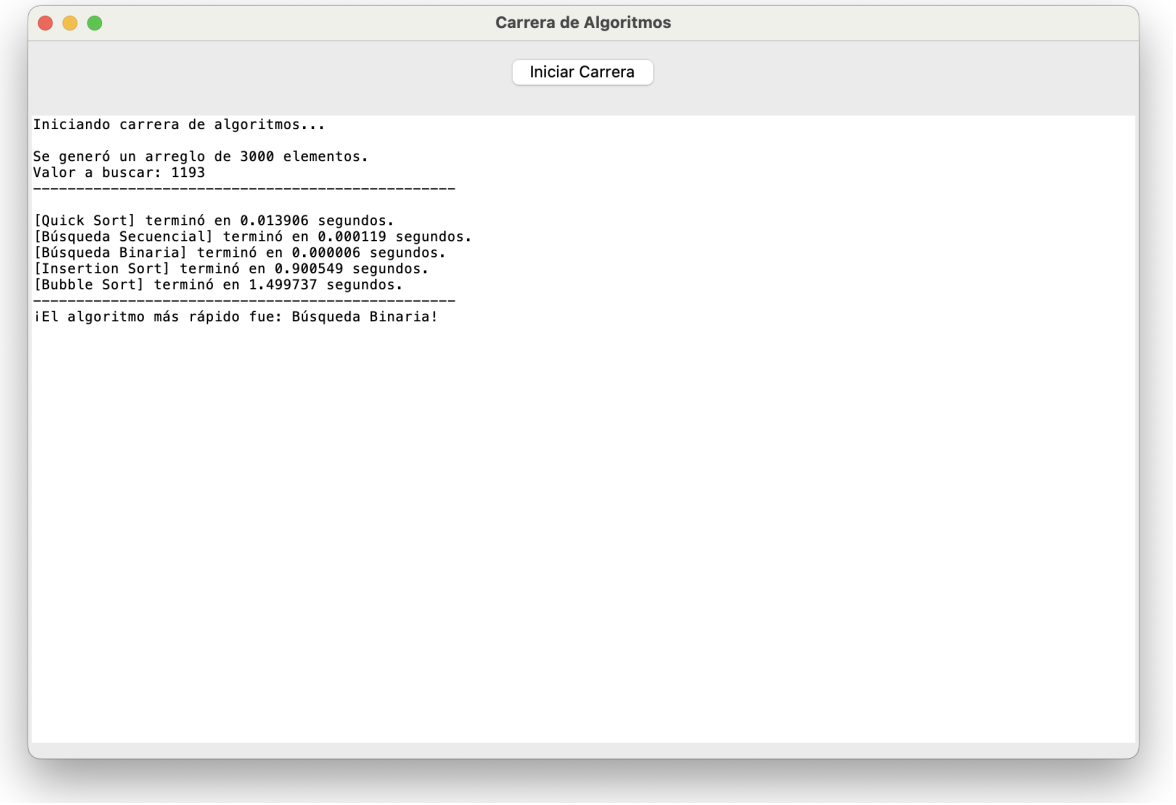
   - Se crea un arreglo aleatorio de tamaño `N` (ej. 3000 elementos).
   - Se selecciona un valor al azar para ser buscado.
   - Se instancian cinco hilos, uno por cada algoritmo:
     - Bubble Sort
     - Quick Sort
     - Insertion Sort

- Búsqueda Secuencial
- Búsqueda Binaria (usando el arreglo ordenado para su búsqueda)

- Cada hilo inicia su procesamiento y mide su tiempo de ejecución con la función `time.perf_counter()`.
- Al terminar, cada hilo reporta su tiempo a la interfaz.
- Finalmente, el sistema muestra cuál algoritmo terminó primero y los tiempos de cada uno.

2. Fotos de la aplicación





3. **Link de Github y Ejecutable de la aplicación**

   https://github.com/RutsSantos/proyecto-final-algoritmos-paralelos

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

   - Quick sort = 0.013906 seg
   - Búsqueda secuencial = 0.000119 seg
   - Busqueda binaria = 0.000006 seg
   - Insertion sort = 0.900549 seg
   - Bubble sort = 1.499757 seg

5. **¿Qué tanta memoria se consumió este proceso?**

   En una medición aproximada (usando herramientas del sistema o módulos como tracemalloc), el uso de memoria no es significativo para 3000-5000 elementos, generalmente algunos megabytes, dependiendo del sistema.

   Para aplicaciones más grandes (p.ej., arreglos de millones de elementos), se requeriría más memoria, especialmente al tener múltiples copias de los datos.

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

   - **Búsquedas:**

     - *Búsqueda Secuencial* recorre todo el arreglo en el peor de los casos (O(n)).

     - *Búsqueda Binaria* funciona en O(log n), pero requiere previamente un arreglo ordenado.

   - **Ordenamientos:**

     - *Bubble Sort* tiene complejidad $O(n^2)$.

     - *Quick Sort* tiene en promedio O(n log n), aunque en el peor caso también puede ser $O(n^2)$.

     - *Método de Inserción* también presenta $O(n^2)$ en el peor caso.

   En general, Quick Sort suele ser el más rápido para la mayoría de arreglos grandes, mientras que la Búsqueda Binaria es significativamente rápida (O(log n)) cuando se compara con la Búsqueda Secuencial; sin embargo, esto asume que el costo de ordenar el arreglo ya fue pagado por algún algoritmo de ordenamiento.

Conclusión

Mediante la ejecución concurrente de los algoritmos de búsqueda y ordenamiento, se observa la diferencia de tiempos de ejecución y la eficiencia de cada técnica. Quick Sort típicamente es el ganador entre los métodos de ordenamiento comparados, mientras que la Búsqueda Binaria supera con creces a la Secuencial siempre que se disponga de un arreglo ordenado.

La aplicación permite evidenciar la importancia de elegir el algoritmo correcto según las necesidades, la naturaleza de los datos y la disponibilidad de recursos. Además, aunque la implementación aquí no es un paralelo "puro", la metodología ejemplifica la idea de ejecución simultánea de procesos para comparar rendimiento o simular un escenario multiproceso.

Bibliografías

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

Documentación oficial de Python (para `threading` y `tkinter`): https://docs.python.org/3/library/

Knuth, D. E. (1998). *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Addison-Wesley.