**UNIVERSIDAD NACIONAL DE LANÚS**

**Trabajo Práctico**

Comparación de Algoritmo QuickSort Secuencial y Concurrente

Materia: Programación Concurrente  
Profesor: Nicolás Pérez  
Alumno: Lucas La Rosa  
DNI: 41.465.362

Lanús, Buenos Aires  
Mayo de 2025

**CONCURRENCIA CON QUICKSORT**

**RESUMEN**

En el siguiente documento se desarrollará como dinámica de trabajo el análisis del algoritmo identificado como Quicksort, en primera instancia se dará una presentación formal del algoritmo y una explicación breve de su funcionamiento.

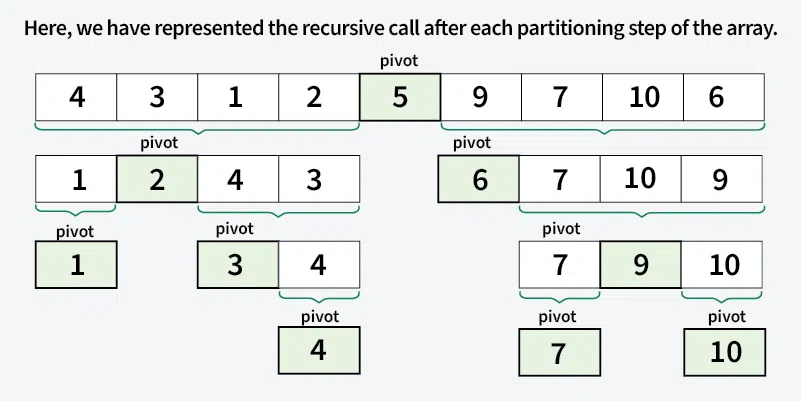
Ondaremos en la mayoría de sus características enfocándonos en virtudes y problemáticas del mismo, como así también la aplicación e incidencia que conlleva su uso para la implementación de concurrencia en la programación. Intentaremos aplicar en todo el transcurso del texto gráficos y código sencillo para hacer amena la comprensión del lector.

# **INTRODUCCIÓN**

Quicksort se basa en un enfoque que busca particionar / dividir el problema original en varios más sencillos, para ello selecciona un elemento como estructura de comparación para con los demás usualmente suele definírselo como “pivote”. Se implementa una subdivisión entre los elementos de menor valor y mayor valor comparados con el pivote, es decir entonces que a cada lado del pivote se genera una sublista con elementos ello se ejecutara en un tiempo lineal ( es decir sin concurrencia ).

Es necesario tener en cuenta que la elección del pivote es de extrema importancia, lo ideal sería poder reconocer el central ya que de esta forma se divide en dos sublistas iguales sin embargo muchas veces ello resulta difícil y demanda demasiado tiempo, motivo por el cual se suele estilar elegir el primero , el último , la media o uno aleatorio.

A continuación, podrá observar una imagen ilustrativa del funcionamiento:



Como se observa en la imagen este proceso se aplica de forma recurrente hasta terminar con el ordenamiento.

**IMPLEMENTACIÓN CONCURRENTE**

Por consiguiente, en la implementación de paralelismo y el uso del denominado multi-threading, en distintas documentaciones observamos que para un algoritmo que se basa en la premisa de dividir y trabajar con problemas más pequeños la idea de utilizar el framework de forkjoinpool resulta bastante conveniente.

En primera medida debemos explicar que es forkjoinpool,se trata de una clase de java que forma parte del framework de concurrencia que se introdujo en Java 7 con la visión de sacarle provecho a los procesadores multinúcleo.

El mismo consta de implementar un patrón fork/join.

“Fork” busca dividir las tareas en otras más pequeñas teniendo en cuenta siempre algún tipo de límite para la subdivisión y por su parte “Join” apunta a esperar que dichas subtareas finalicen y combinar los resultados de las mismas. -

Es así que además converge en este clase otra muy importante conocida como forkjointask, una clase abstracta que tiene como objetivo representar la tarea a ejecutar la misma extiende de recursivetask (si devuelve un valor ) o recursiveaction (si no devuelve resultado).

Como ultimo comentario se debe recordar que el método “compute” debe ser sobrescrito de manera obligatoria, en dicho método se llevara a cabo lo esencial del algoritmo recursivo o el algoritmo secuencial según se cumpla o no la condición.

Teniendo en cuente este efímero paneo procedemos a analizar el siguiente código**:**

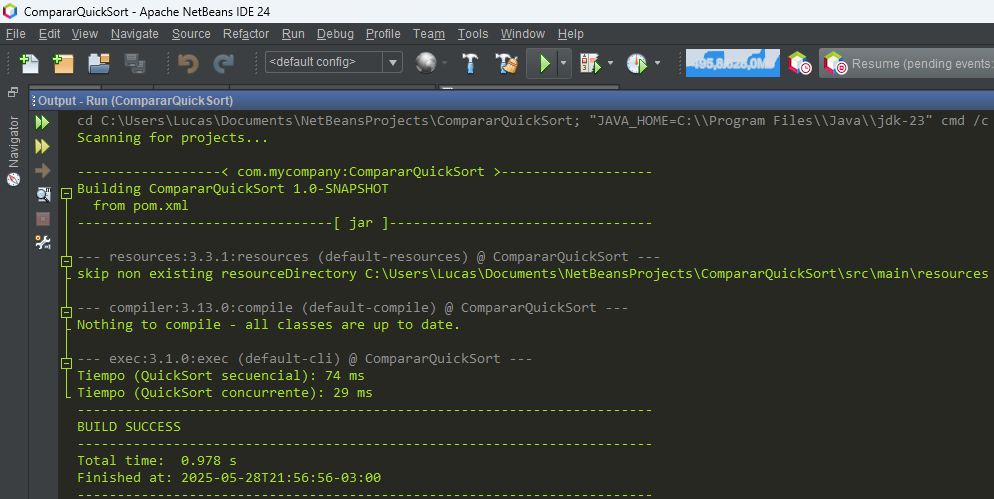
|  |
| --- |
| import java.util.Arrays;  import java.util.Random;  import java.util.concurrent.RecursiveAction;  import java.util.concurrent.ForkJoinPool;  public class CompararQuickSort {  **// ------------------VERSIÓN SECUENCIAL -----------------------**  public static void quickSortSecuencial(int[] arr, int inicio, int fin) {  if (inicio < fin) {  int pivote = particionar(arr, inicio, fin);  quickSortSecuencial(arr, inicio, pivote - 1);  quickSortSecuencial(arr, pivote + 1, fin);  }  }  **//------------- VERSIÓN CONCURRENTE ----------------------------**  public static class QuickSortForkJoin extends RecursiveAction {  private int[] arr;  private int inicio, fin;  private static final int UMBRAL = 1000;  **//---------------------CONSTRUCTOR-------------------------------**  public QuickSortForkJoin(int[] arr, int inicio, int fin) {  this.arr = arr;  this.inicio = inicio;  this.fin = fin;  }  @Override  **// ---------------------METODO QUE SE SOBREESCRIBE----------------**  protected void compute() {  if (inicio < fin) {  if (fin - inicio < UMBRAL) {  quickSortSecuencial(arr, inicio, fin);  } else {  int pivote = particionar(arr, inicio, fin);  invokeAll(  new QuickSortForkJoin(arr, inicio, pivote - 1),  new QuickSortForkJoin(arr, pivote + 1, fin));  }}}}  **// -------------------- METODO DE PARTICION --------------------**  public static int particionar(int[] arr, int inicio, int fin) {  int pivote = arr[inicio];  int i = inicio + 1;  int j = fin;  while (i <= j) {  while (i <= j && arr[i] <= pivote) i++;  while (i <= j && arr[j] > pivote) j--;  if (i < j) {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int temp = arr[inicio];  arr[inicio] = arr[j];  arr[j] = temp;  return j;}    **// -----------------------MAIN ------------------------------**  public static void main(String[] args) {  int[] original = new Random().ints(1\_000\_000, 0, 1\_000\_000).toArray(); **// Array aleatorio**  **// COPIAS PARA QUE AMBOS MÉTODOS TRABAJEN CON LOS MISMOS DATOS**  int[] copiaSecuencial = Arrays.copyOf(original, original.length);  int[] copiaConcurrente = Arrays.copyOf(original, original.length);  **// ------------------ MEDIR TIEMPO SECUENCIAL-------------------**  long inicioSec = System.nanoTime();  quickSortSecuencial(copiaSecuencial, 0, copiaSecuencial.length - 1);  long finSec = System.nanoTime();  long tiempoSec = (finSec - inicioSec) / 1\_000\_000;  System.out.println("Tiempo (QuickSort secuencial): " + tiempoSec + " ms");  **// -------------- MEDIR TIEMPO CONCURRENTE--------------------------**  long inicioConc = System.nanoTime();  ForkJoinPool pool = new ForkJoinPool();  pool.invoke(new QuickSortForkJoin(copiaConcurrente, 0, copiaConcurrente.length - 1));  long finConc = System.nanoTime();  long tiempoConc = (finConc - inicioConc) / 1\_000\_000;  System.out.println("Tiempo (QuickSort concurrente): " + tiempoConc + " ms");  }  } |

**COMPARATIVA**

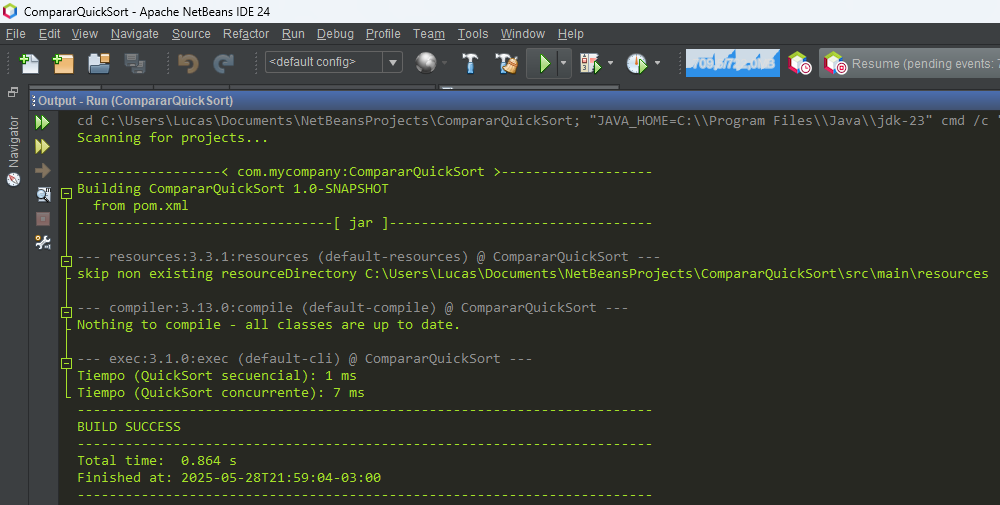
En esta sección de nuestro análisis, dejaremos plasmadas las distintas pruebas efectuadas en los puntos bisagra del algoritmo, es decir aquellos que tienen incidencia directamente con el buen o mal funcionamiento de la concurrencia en base a las elecciones tomadas.

Test de eficiencia con array desordenado:

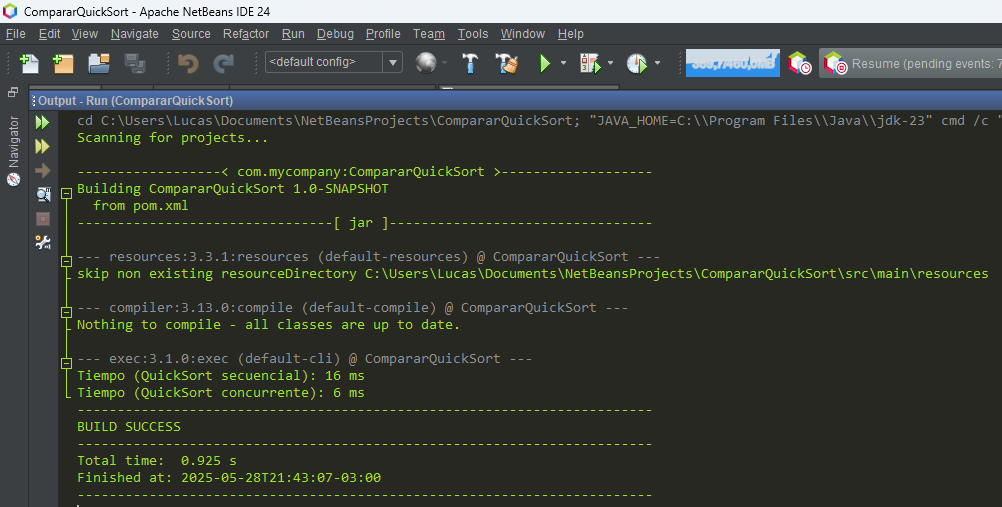
**Umbral 10 Ramdom 1000000:**



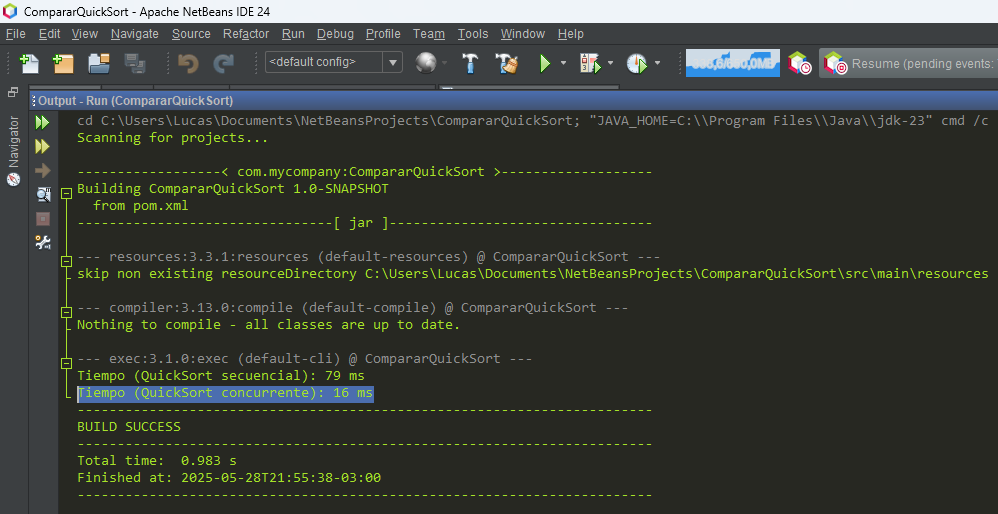
**Umbral 10 Ramdom 10000:**



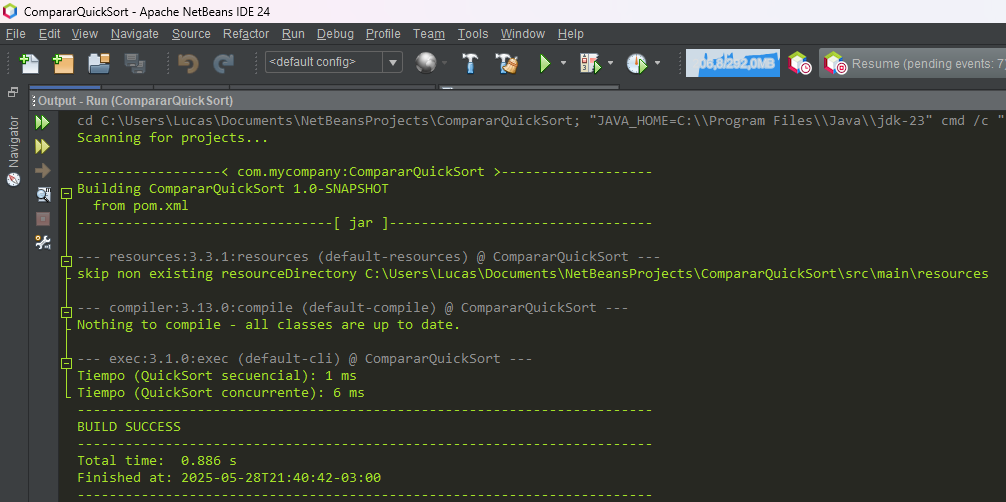
**Umbral 100 Ramdom 100000:**



**Umbral 100 Ramdom 1000000:**



**Umbral 100 Ramdom 10000:**

****

En base a las pruebas realizadas se observa que el método de Quicksort aplicando concurrencia es muy efectivo cuando se trabaja con una gran cantidad de elementos mayores a 100.000 con umbrales de 10 y 100 en este caso el Big O aplica O (n log n), reduciendo el tiempo de ejecución a mas de un 50%.

**DESCRIPCIÓN DEL EQUIPO UTILIZADO:**

Ryzen 7 5700g con 8 núcleos y 16 hilos

16 GB de Ram

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N°De Prueba | Umbral | Cant. De elementos | Tiempo Secuencial | Tiempo Concurrente |
| 1 | 10 | 1000000 | 74 ms | 29ms |
| 2 | 10 | 1000 | 1 ms | 7 ms |
| 3 | 100 | 100000 | 16ms | 6ms |
| 4 | 100 | 1000000 | 79ms | 16ms |
| 5 | 100 | 10000 | 1ms | 6ms |

**CONCLUSION**

Como análisis final observamos que las versiones secuencial y concurrente del algoritmo QuickSort, tienen notables diferencias de rendimiento. Para volúmenes de datos grandes (mayores a 100.000 elementos) es recomendable utilizar concurrencia. No obstante, en datasets (colección de datos) pequeños, la versión secuencial resulta más rápida debido al overhead (sobrecarga) que implica la gestión de hilos, es decir la creación y destrucción de hilos, la administración del ForkJoinPool o estructura paralela que divide y distribuye tareas y la sincronización entre hilos y el "merge" (combinación) de resultados. Todo ello **requiere tiempo que se pierde en esta gestión y al fin de cuentas puede ser mayor que el tiempo necesario para ordenar.**

El umbral de partición juega un rol fundamental en este equilibrio: un valor adecuado (como 100) permite aprovechar la concurrencia sin incurrir en sobrecargas innecesarias. En síntesis, la implementación concurrente es recomendable cuando se trabaja con grandes cantidades de datos y se cuenta con procesadores multinúcleo; para arreglos pequeños, la versión secuencial es más eficiente y simple.

Esta comparación permite concluir que la elección del enfoque (secuencial o el concurrente) debe considerar tanto el tamaño de los datos con los cuales se está trabajando como así también la arquitectura del hardware en donde se ejecuta el programa, ello en pos de que ForkJoinPool utiliza múltiples núcleos del CPU.-

**REFERENCIAS**

Baeldung. (2022, julio 15). QuickSort in Java. <https://www.baeldung.com/java-quicksort>

Chio Code. (2022, septiembre 6). Java | 77. Algoritmo QuickSort (Método de ordenamiento) [Video]. YouTube. <https://www.youtube.com/watch?v=UrPJLhKF1jY&t=39s>

GeeksforGeeks. (2023, abril 5). QuickSort in Java. <https://www.geeksforgeeks.org/java-program-for-quicksort/>

GeeksforGeeks. (2023, septiembre 14). Quick Sort using Multi-threading. <https://www.geeksforgeeks.org/quick-sort-using-multi-threading/>

La Rosa, L. (2025). *Análisis de concurrencia y secuencialidad: QuickSort* [Repositorio GitHub]. GitHub. <https://github.com/WOLF2235/Analisis-concurrencia-y-secuencialidad-Quicksort->

La Rosa, L. (2025). *QUICKSORT VIDEO* [Archivo de video]. GitHub. <https://github.com/WOLF2235/Analisis-concurrencia-y-secuencialidad-Quicksort-/blob/468569b05f62004a30d3fd6778720952fda779c1/QUICKSORT%20VIDEO.mp4>