# Proyecto de Análisis de Algoritmos de ordenamiento

Abigail Becerra

David Jacome

Edison Ordoñez

Escuela Politécnica Nacional

Estructura de Datos y Algoritmos I

Ing. Boris Astudillo

12/11/2025

Índice



## 1. Introducción

El trabajo se centra en un archivo compuesto por un millón de registros que representan información sobre diferentes películas. Este tipo de dataset será utilizada en un entorno en el cual se lleva a cabo un análisis de los diferentes atributos que tiene la tabla o base de datos, el cual permite aplicar distintos métodos estadísticos, comparativos y computacionales de los diferentes atributos de ordenamiento.

### 1.1. Objetivo

Desarrollar un experimento computacional de un conjunto de datos de un millón de registros, implementando dos algoritmos de ordenamiento diferentes con contadores de

tiempo para medir el rendimiento de cada uno, y aplicando técnicas de multiprocesamiento en Python en diferentes equipos para comparar y analizar la eficiencia del procesamiento de Big Data bajo diferentes condiciones de hardware.

## 2. Marco Teórico

### 2.1. Base de Datos utilizada para el proyecto

El proyecto en si debe permitirnos ordenar una base de datos de gran tamaño, en este caso tiene más de un millón de datos. La base de datos es de información recopilada de cintas cinematográficas, a las cuales tenemos atributos como su id, titulo, calificación promedio, fecha de estreno, presupuesto y la popularidad. En esta base de datos usaremos dos diferentes algoritmos de ordenamiento, el primero será de MergeSort y el segundo de QuickSort, después evaluaremos el tiempo que se tardó en ejecutar cada uno en diferentes equipos y haremos comparaciones.

### 2.2. La Fórmula que se usará

La a fórmula utilizada en este proyecto no se basa en datos reales de impuestos ni variaciones presupuestarias, sino en una simulación aleatoria que permite modelar escenarios económicos diversos para cada película. Esta simulación se compone de tres elementos principales.

Variación

La variación representa un incremento simulado en el presupuesto original de cada película. Este aumento se genera aleatoriamente dentro de un rango del 5% al 15%,

con el objetivo de reflejar posibles ajustes financieros como gastos adicionales, inflación o cambios en la producción.

Impuesto

El impuesto simulado corresponde a una carga financiera o descuento aplicado sobre el presupuesto ajustado. Se asigna aleatoriamente un valor de 5%, 10% o 15% a cada película, lo que permite representar distintos escenarios fiscales o comerciales

Valor Final

El valor final es el resultado de aplicar la variación y el impuesto al presupuesto original. Este valor representa el presupuesto ajustado neto de cada película, y se utiliza como criterio principal para el ordenamiento.

$$Valor\ Final\ =\ Budget\ \cdot\ (1\ +\ Variación)\ \cdot\ (1-Impuesto)$$

## 2.3. Algoritmos de Ordenamientos de los que se hará uso

MergeSort

Es un algoritmo que divide recursivamente la lista en mitades hasta que cada sublista contiene un solo elemento. Luego, estas sublistas se combinan en orden, formando listas cada vez más grandes hasta reconstruir la lista original, pero ordenada.

QuickSort

también divide la lista, pero lo hace eligiendo un elemento llamado pivote. Luego reorganiza los elementos de forma que todos los menores al pivote queden a su izquierda y los mayores a su derecha. Este proceso se repite recursivamente en cada sublista.

## 3. Metodología

Dentro de este proyecto, se hizo uso de 3 equipos computacionales con las siguientes especificaciones cada uno.

| Especificaciones | Abigail | David | Edison |
|---|---|---|---|
| **Procesador** | Intel(R) Core i7-1265U (12ª generación) | Intel(R) Core i7-1355U (13ª generación) | Intel(R) Pentium(R) CPU |
| **Frecuencia del Procesador** | 1.80 GHz | 1.70GHz | 2.16 GHz |
| **Núcleos del Procesador** | 10 | 10 | 4 |
| **RAM** | 16 | 16 | 8 |

TABLA 1. ESPECIFICACIONES DE EQUIPOS

### 3.1. Ejecución en Python

- Importación de librerias necesarias del proyecto.

```
import pandas as pd
import numpy as np
import time
import warnings
from multiprocessing import Pool, cpu_count

warnings.filterwarnings("ignore", category=FutureWarning)
```

- Lectura del archivo .csv en el proyecto.

```
ruta = "BasedeDatosMoviesEDA.csv"

for enc in ["utf-8-sig", "cp1252", "latin-1", "ISO-8859-1"]:
    try:
        df = pd.read_csv(
            ruta,
            engine="python",
            sep=None,
            on_bad_lines="skip",
            encoding=enc
        )
        print("Cargó con encoding:", enc)
        break
    except UnicodeDecodeError as e:
        print("Falló con", enc, "->", e)

# Mostrar información del DataFrame
print(df.shape)
display(df.head())
print(df.info())
```

- Fórmulas que podrían usarse para la database.

```
import numpy as np

# Se simula una variación del 5% a 15%
df["variacion"] = (df["budget"] * (1 + np.random.uniform(0.05, 0.15, len(df)))).round(2)

# Se simula un impuesto o descuento
df["impuesto"] = np.random.choice([0.05, 0.10, 0.15], size=len(df)).round(2)

# Se calcula un valor final
df["valor_final"] = (df["variacion"] * (1 - df["impuesto"])).round(2)

# Ordenar por valor_final de mayor a menor
df = df.sort_values(by="valor_final", ascending=False)

# Imprimir el DataFrame resultante
print(df[["title", "budget", "variacion", "impuesto", "valor_final"]].head(10))
```

- Método del Algoritmo de Ordenamiento de MergeSort.

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i][1] <= right[j][1]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

- Método del Algoritmo de Ordenamiento de QuickSort.

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2][1]
    left = [x for x in arr if x[1] < pivot]
    middle = [x for x in arr if x[1] == pivot]
    right = [x for x in arr if x[1] > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

- Método para ejecutar los algoritmos.

```python
def ordenar_dataframe(df, algoritmo):
    """Ordena un DataFrame según 'valor_final' usando el algoritmo dado."""
    arr = list(zip(df.index, df["valor_final"].values))
    inicio = time.perf_counter()
    if algoritmo == "mergesort":
        ordenado = merge_sort(arr)
    elif algoritmo == "quicksort":
        ordenado = quick_sort(arr)
    else:
        raise ValueError("Algoritmo no reconocido. Usa 'mergesort' o 'quicksort'.")
    fin = time.perf_counter()

    indices_ordenados = [i for i, _ in ordenado]
    df_ordenado = df.loc[indices_ordenados].reset_index(drop=True)
    return df_ordenado, fin - inicio
```

- Aplicación de los métodos en la base de datos y tiempo que se tardaron en ejecutar cada algoritmo.

```python
print("\n--- ORDENAMIENTO SIN MULTIPROCESAMIENTO ---")

df_merge, tiempo_merge = ordenar_dataframe(df, "mergesort")
print(f" MergeSort completado en {tiempo_merge:.3f} segundos")

df_quick, tiempo_quick = ordenar_dataframe(df, "quicksort")
print(f" QuickSort completado en {tiempo_quick:.3f} segundos")

print("-------------------------------------------------")

if tiempo_merge < tiempo_quick:
    print(f" El algoritmo más rápido fue: MergeSort ({tiempo_merge:.3f}s)")
else:
    print(f" El algoritmo más rápido fue: QuickSort ({tiempo_quick:.3f}s)")
```

- Métodos para iniciar el multiprocesamiento, nuevamente estarán los algoritmos, pero ya integrados al multiprocesamiento.

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

def worker_mergesort(chunk):
    arr = list(zip(chunk.index, chunk["valor_final"].values))
    ordenado = merge_sort(arr)
    indices_ordenados = [i for i, _ in ordenado]
    return chunk.loc[indices_ordenados]

def worker_quicksort(chunk):
    arr = list(zip(chunk.index, chunk["valor_final"].values))
    ordenado = quick_sort(arr)
    indices_ordenados = [i for i, _ in ordenado]
    return chunk.loc[indices_ordenados]

def ejecutar_multiproceso(df, algoritmo):
    num_procesos = min(4, cpu_count())
    # repartir todas las filas, sin perder sobrantes
    chunks = [c for c in np.array_split(df, num_procesos) if len(c) > 0]

    print(f"\n Iniciando multiprocesamiento ({algoritmo}) con {len(chunks)} hilos...")
    t0 = time.perf_counter()

    if algoritmo == "mergesort":
        worker_func = worker_mergesort
    elif algoritmo == "quicksort":
        worker_func = worker_quicksort
    else:
        raise ValueError("Algoritmo inválido.")

    with ThreadPoolExecutor(max_workers=len(chunks)) as executor:
        futures = [executor.submit(worker_func, chunk) for chunk in chunks]
        partes = [future.result() for future in as_completed(futures)]

    # concatenar todas las partes; reset_index si quieres índice secuencial
    df_final = pd.concat(partes).reset_index(drop=True)
    t1 = time.perf_counter()

    tiempo_total = t1 - t0
    print(f" {algoritmo.capitalize()} paralelo completado en {tiempo_total:.3f} segundos.")
    return df_final, tiempo_total
```

- Aplicación del multiprocesamiento y verificación de la duración de este en cada uno de los algoritmos.

```python
print("\n--- ORDENAMIENTO CON MULTIPROCESAMIENTO ---")

df_merge_multi, tiempo_merge_multi = ejecutar_multiproceso(df, "mergesort")
print(f" MergeSort paralelo completado en {tiempo_merge_multi:.3f} segundos")

df_quick_multi, tiempo_quick_multi = ejecutar_multiproceso(df, "quicksort")
print(f" QuickSort paralelo completado en {tiempo_quick_multi:.3f} segundos")
```

- Comparación de rendimientos entre procesamiento mononúcleo y multiprocesamiento.

```
print("-------------------------------------------------")
print("\n--- COMPARACIÓN DE RENDIMIENTO ---")
print(f"MergeSort (sin multiproceso): {tiempo_merge:.3f}s")
print(f"MergeSort (con multiproceso): {tiempo_merge_multi:.3f}s")
print(f"Mejora: {((tiempo_merge - tiempo_merge_multi) / tiempo_merge * 100):.2f}%")

print(f"\nQuickSort (sin multiproceso): {tiempo_quick:.3f}s")
print(f"QuickSort (con multiproceso): {tiempo_quick_multi:.3f}s")
print(f"Mejora: {((tiempo_quick - tiempo_quick_multi) / tiempo_quick * 100):.2f}%")
```

## 3.2. Ejecución en JAVA

- Importación de librerias necesarias del proyecto.

```
import java.io.*;    ProyectoBimestral.java is a non-pr
import java.util.*;
import java.nio.file.*;
import java.util.concurrent.ThreadLocalRandom;
```

- Lectura del archivo .csv en el proyecto.

```
//Lector de archivo
public static double[] readBudgets(String path, int minSize) throws Exception {
    BufferedReader br = Files.newBufferedReader(Paths.get(path), java.nio.charset.StandardCharsets.ISO_8859_1);
    String header = br.readLine();
    boolean hasHeader = header != null && header.toLowerCase().contains(s: "budget");
    int budgetCol = -1;
    List<Double> vals = new ArrayList<>();
    if (hasHeader) {
        String[] cols = header.split(regex: "[,;]");
        for (int i = 0; i < cols.length; i++) if (cols[i].trim().toLowerCase().equals(anObject: "budget")) budgetCol = i;
    } else {
        budgetCol = 4;
        br = Files.newBufferedReader(Paths.get(path));
    }
    String line;
    while ((line = br.readLine()) != null) {
        String[] parts = line.split(regex: "[,;]");
        if (budgetCol >= parts.length) continue;
        try {
            double b = Double.parseDouble(parts[budgetCol].replaceAll(regex: "[^0-9.\\-]", replacement: ""));
            vals.add(b);
        } catch (Exception e) { /* saltar */ }
        if (vals.size() >= minSize) break;
    }
    // Aplicar valores random
    while (vals.size() < minSize) vals.add( ThreadLocalRandom.current().nextDouble(origin: 0, bound: 1e7) );
    double[] arr = new double[vals.size()];
    for (int i = 0; i < vals.size(); i++) arr[i] = vals.get(i);
    return arr;
}
```

- Fórmulas que podrían usarse para la database.

```java
// Aplicar fórmula: variacion = budget * (1 + r) , impuesto in {0.05,0.10,0.15}, valor_final = variacion*(1-impuesto)
double[] valorFinal = new double[budgets.length];
Random rnd = new Random(seed: 12345);
double[] impuestos = {0.05,0.10,0.15};
for (int i=0;i<budgets.length;i++){
    double r = 0.05 + rnd.nextDouble()*(0.10); // [0.05,0.15)
    double variacion = budgets[i] * (1.0 + r);
    double imp = impuestos[rnd.nextInt(impuestos.length)];
    valorFinal[i] = variacion * (1.0 - imp);
}
```

- Método del Algoritmo de Ordenamiento de MergeSort.

```java
// MergeSort
public static void mergeSort(double[] a, double[] aux, int lo, int hi) {
    if (hi - lo <= 0) return;
    int mid = (lo + hi) >>> 1;
    mergeSort(a, aux, lo, mid);
    mergeSort(a, aux, mid+1, hi);
    int i = lo, j = mid+1, k = lo;
    while (i <= mid || j <= hi) {
        if (j > hi || (i <= mid && a[i] <= a[j])) aux[k++] = a[i++];
        else aux[k++] = a[j++];
    }
    System.arraycopy(aux, lo, a, lo, hi - lo + 1);
}
```

- Método del Algoritmo de Ordenamiento de QuickSort.

```java
// QuickSort
public static void quickSort(double[] a, int lo, int hi) {
    if (lo >= hi) return;
    int i = lo, j = hi;
    double pivot = a[lo + (hi - lo) / 2];
    while (i <= j) {
        while (a[i] < pivot) i++;
        while (a[j] > pivot) j--;
        if (i <= j) {
            double t = a[i]; a[i] = a[j]; a[j] = t;
            i++; j--;
        }
    }
    if (lo < j) quickSort(a, lo, j);
    if (i < hi) quickSort(a, i, hi);
}
```

- Aplicación de los métodos en la base de datos y tiempo que se tardaron en ejecutar cada algoritmo.

```java
// Tiempo Merge Sort
long t0 = System.nanoTime();
mergeSort(aForMerge, aux, lo: 0, aForMerge.length-1);
long t1 = System.nanoTime();
double mergeSec = (t1 - t0) / 1e9;
System.out.printf(format: "Mergesort: %.6f s%n", mergeSec);

// Tiempo Quick Sort
t0 = System.nanoTime();
quickSort(aForQuick, lo: 0, aForQuick.length-1);
t1 = System.nanoTime();
double quickSec = (t1 - t0) / 1e9;
System.out.printf(format: "Quicksort: %.6f s%n", quickSec);

System.out.printf(format: "Mejor ordenamiento: %s%n", (mergeSec < quickSec) ? "Mergesort" : "Quicksort");
```

## 4. Resultados

Como la ejecucion de este programa se lo realizo en 3 diferentes equipos, a continuación, se mostrará los diferentes resultados que dio el programa en cada una de las diferentes computadoras.

### 4.1. Análisis en Python

Se realizo la ejecución sin el multiprocesamiento y los resultados fueron los siguientes:

| Tipo de Algoritmo | Abigail | David | Edison |
|---|---|---|---|
| **MergeSort** | 1.336 Seg | 2.872 Seg | 9.993 Seg |
| **QuickSort** | 0.191 Seg | 0.377 Seg | 1.094 Seg |

TABLA 2. EJECUCION SIN MULTIPROCESAMIENTO EN PYTHON

Y estas fueron con el multiprocesamiento:

| Tipo de Algoritmo | Abigail | David | Edison |
|---|---|---|---|
| **MergeSort** | 2.021 Seg | 3.505 Seg | 26.568 Seg |
| **QuickSort** | 1.108 Seg | 1.136 Seg | 7.385 Seg |

TABLA 3. EJECUCION CON MULTIPROCESAMIENTO EN PYTHON

### 4.2. Análisis en JAVA

Se realizo la ejecución sin el multiprocesamiento y los resultados fueron los siguientes:

| Tipo de Algoritmo | Abigail | David | Edison |
|---|---|---|---|
| **MergeSort** | 0.118 Seg | 0.0632 Seg | 0.261 Seg |
| **QuickSort** | 0.088 Seg | 0.0605 Seg | 0.166 Seg |

TABLA 4. EJECUCION SIN MULTIPROCESAMIENTO EN JAVA

## 5. Análisis y Discusión

A partir de los tiempos de ejecución registrados en las pruebas realizadas con el conjunto de datos del proyecto, donde se evaluaron MergeSort y QuickSort tanto en modo mononúcleo como en paralelo. En las ejecuciones sin multiprocesamiento QuickSort mostró un desempeño notablemente superior al de MergeSort, tomando de ejemplo al equipo más lento, el algoritmo de ordenamiento termino en aproximadamente 1.094 segundos frente a los 9.993 segundos reportados para MergeSort. No obstante, al introducir multiprocesamiento los resultados cambian de forma inesperada: la versión paralela de MergeSort aumentó su tiempo a 26.568 segundos y la versión paralela de QuickSort a 7.385 segundos, evidenciando que en este experimento el paralelismo no produjo la mejora esperada y, en el caso de MergeSort, incluso degradó significativamente el rendimiento. Por lo tanto, los valores observados indican que para el tamaño y la estrategia de particionado empleada, el paralelismo introdujo más overhead que beneficio.

## 6. Conclusiones

La culminación de este proyecto muestra una aplicación práctica de algoritmos de ordenamiento sobre un conjunto de datos cinematográficos de gran tamaño, integrando además una simulación económica basada en variación presupuestaria e impuestos aleatorios para generar un indicador financiero (valor_final). Los resultados muestran que QuickSort ofrece un rendimiento superior en ejecución mononúcleo para el atributo seleccionado, mientras que MergeSort.

La experimentación con multiprocesamiento reveló que el paralelismo, tal como fue implementado en este trabajo, no aseguró mejoras en si: las versiones paralelas incrementaron el tiempo total en nuestros ensayos, indicando que los costes de particionado, comunicación y copia de fragmentos de DataFrame pueden superar las ventajas de la concurrencia. Esto subraya la necesidad de un diseño cuidadoso del paralelismo para evitar deficiencias dentro de bases de datos empresariales.

En conjunto, el proyecto valida la importancia de seleccionar el algoritmo y la estrategia de paralelización adecuados según el contexto, y proporciona una base reproducible para optimizaciones futuras y estudios más amplios con herramientas orientadas a Big Data.

## 7. Bibliografía

Auger, N., Nicaud, C., & Pivoteau, C. (2015). Merge strategies: from merge sort to Timsort.

Sedgewick, R. (1978). Implementing quicksort programs. Communications of the ACM, 21(10), 847-857.

Sedgewick, R. (1978). Implementing quicksort programs. Communications of the ACM, 21(10), 847-857.

Aziz, Z. (2021). Python parallel processing and multiprocessing: A rivew. Academic Journal of Nawroz University.