Estructuras de datos

AI07064821 Diego Villarreal
AI07050780 Ricardo Delgado

Proyecto final

Profesor: Francisco Gomez Rubio

Source code

https://github.com/WebbiestStew/proyectofinal-ed.git

Objetivo

Nuestro objetivo a lo largo del proyecto fue que sea una programación concurrente y comprensión de algoritmos de ordenamiento mediante una implementación y ordenación de seis métodos de ordenamiento (se mencionan después de estas introducciones)

¿Qué hace el proyecto?

Se comparan 6 algoritmos de ordenamiento ejecutándose **simultáneamente** y mide cuántas colecciones puede ordenar cada uno en tiempo límite establecido por el usuario

Implementación de los 6 métodos de ordenamiento

Archivo: AlgoritmosOrdenamiento.java
Este mismo contiene los 6 métodos de ordenamiento que se nos indicó

- BubbleSort(): compara los elementos adyacentes y los intercambia
- InsertionSort(): inserta cada elemento en su posición correcta
- SelectionSort(): selecciona el mínimo y lo coloca al inicio
- MergeSort(): divide el arreglo y luego mezcla ordenadamente
- QuickSort(): Usa un “pivote” para dividir y ordenar
- HeapSort(): Construye un “heap” y extrae elementos ordenados

Codigo a continuacion:
BubbleSort():

InsertionSort():

```java

    public static int[] insertionSort(int[] arreglo) {
        int[] resultado = clonarArreglo(arreglo);
        int n = resultado.length;

        for (int i = 1; i < n; i++) {
            int clave = resultado[i];
            int j = i - 1;

            while (j >= 0 && resultado[j] > clave) {
                resultado[j + 1] = resultado[j];
                j--;
            }

            resultado[j + 1] = clave;
        }

        return resultado;
    }

```

SelectionSort():

```java

    public static int[] selectionSort(int[] arreglo) {
        int[] resultado = clonarArreglo(arreglo);
        int n = resultado.length;

        for (int i = 0; i < n - 1; i++) {
            int indiceMinimo = i;

            for (int j = i + 1; j < n; j++) {
                if (resultado[j] < resultado[indiceMinimo]) {
                    indiceMinimo = j;
                }
            }

            int temp = resultado[indiceMinimo];
            resultado[indiceMinimo] = resultado[i];
            resultado[i] = temp;
        }

        return resultado;
    }
```

MergeSort():

```java

    public static int[] mergeSort(int[] arreglo) {
        if (arreglo.length <= 1) {
            return clonarArreglo(arreglo);
        }

        int[] resultado = clonarArreglo(arreglo);
        mergeSortRecursivo(resultado, izquierda: 0, resultado.length - 1);
        return resultado;
    }

    private static void mergeSortRecursivo(int[] arr, int izquierda, int derecha) {
        if (izquierda < derecha) {
            int medio = izquierda + (derecha - izquierda) / 2;

            mergeSortRecursivo(arr, izquierda, medio);
            mergeSortRecursivo(arr, medio + 1, derecha);

            mezclar(arr, izquierda, medio, derecha);
        }
    }
```

```java

    private static void mezclar(int[] arr, int izquierda, int medio, int derecha) {
        int n1 = medio - izquierda + 1;
        int n2 = derecha - medio;

        int[] izq = new int[n1];
        int[] der = new int[n2];

        for (int i = 0; i < n1; i++) {
            izq[i] = arr[izquierda + i];
        }
        for (int j = 0; j < n2; j++) {
            der[j] = arr[medio + 1 + j];
        }

        int i = 0, j = 0, k = izquierda;

        while (i < n1 && j < n2) {
            if (izq[i] <= der[j]) {
                arr[k] = izq[i];
                i++;
            } else {
                arr[k] = der[j];
                j++;
            }
            k++;
        }
```

QuickSort():

```
120
121     public static int[] quickSort(int[] arreglo) {
122         if (arreglo.length <= 1) {
123             return clonarArreglo(arreglo);
124         }
125
126         int[] resultado = clonarArreglo(arreglo);
127         quickSortRecursivo(resultado, bajo: 0, resultado.length - 1);
128         return resultado;
129     }
130
131     private static void quickSortRecursivo(int[] arr, int bajo, int alto) {
132         if (bajo < alto) {
133             int indicePivote = particionar(arr, bajo, alto);
134
135             quickSortRecursivo(arr, bajo, indicePivote - 1);
136             quickSortRecursivo(arr, indicePivote + 1, alto);
137         }
138     }
139
140     private static int particionar(int[] arr, int bajo, int alto) {
141         int pivote = arr[alto];
142         int i = bajo - 1;
143
144         for (int j = bajo; j < alto; j++) {
145             if (arr[j] < pivote) {
146                 i++;
147                 int temp = arr[i];
148                 arr[i] = arr[j];
149                 arr[j] = temp;
150             }
151         }
152
153         int temp = arr[i + 1];
154         arr[i + 1] = arr[alto];
155         arr[alto] = temp;
156
157         return i + 1;
158     }
```

HeapSort():

```java

    public static int[] heapSort(int[] arreglo) {
        int[] resultado = clonarArreglo(arreglo);
        int n = resultado.length;

        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(resultado, n, i);
        }

        for (int i = n - 1; i > 0; i--) {
            int temp = resultado[0];
            resultado[0] = resultado[i];
            resultado[i] = temp;

            heapify(resultado, i, i: 0);
        }

        return resultado;
    }

    private static void heapify(int[] arr, int n, int i) {
        int mayor = i;
        int izquierdo = 2 * i + 1;
        int derecho = 2 * i + 2;

        if (izquierdo < n && arr[izquierdo] > arr[mayor]) {
            mayor = izquierdo;
        }

        if (derecho < n && arr[derecho] > arr[mayor]) {
            mayor = derecho;
        }

        if (mayor != i) {
            int temp = arr[i];
            arr[i] = arr[mayor];
            arr[mayor] = temp;

            heapify(arr, n, mayor);
        }
    }
```

Un método simple para clonar arreglos:

```
200
201         You, 1 second ago • Uncommitted changes
202     private static int[] clonarArreglo(int[] original) {
203         int[] copia = new int[original.length];
204         for (int i = 0; i < original.length; i++) {
205             copia[i] = original[i];
206         }
207         return copia;
208     }
209 }
210
```

Volúmenes de datos generados aleatoriamente

Archivo: GeneradorColecciones.java
Genera los 4 tipos de colecciones especificados

```java
    public int[] generar100Elementos() {
        return generarAleatorios(cantidad: 100, valorMaximo: 100000);
    }

    public int[] generar50000Elementos() {
        return generarAleatorios(cantidad: 50000, valorMaximo: 100000);
    }

    public int[] generar100000Elementos() {
        return generarAleatorios(cantidad: 100000, valorMaximo: 100000);
    }

    public int[] generar100000Restringidos() {
        return generarAleatorios(cantidad: 100000, valorMaximo: 5);
    }

    private int[] generarAleatorios(int cantidad, int valorMaximo) {
        int[] datos = new int[cantidad];

        for (int i = 0; i < cantidad; i++) {
            semilla = (semilla * 1103515245 + 12345) & 0x7fffffffL;
            datos[i] = ((int)(semilla % valorMaximo)) + 1;
        }

        return datos;
    }
```

```java
    public String obtenerDescripcion(int tipo) {
        switch (tipo) {
            case 1:
                return "100 elementos aleatorios";
            case 2:
                return "50,000 elementos aleatorios";
            case 3:
                return "100,000 elementos aleatorios";
            case 4:
                return "100,000 elementos (1-5)";
            default:
                return "Desconocido";
        }
    }

    public int[] generarPorTipo(int tipo) {
        switch (tipo) {
            case 1:
                return generar100Elementos();
            case 2:
                return generar50000Elementos();
            case 3:
                return generar100000Elementos();
            case 4:
                return generar100000Restringidos();
            default:
                return new int[0];
        }
    }
}
```

En caso de no introducir ninguno, se pone un “desconocido”
NO usa java.util, porque? Si se permite, pero para tener que seguir con el patrón que se usaba de no usar el java.util, no se usó en este proyecto, optamos por un generador LCG (linear congruential generator)

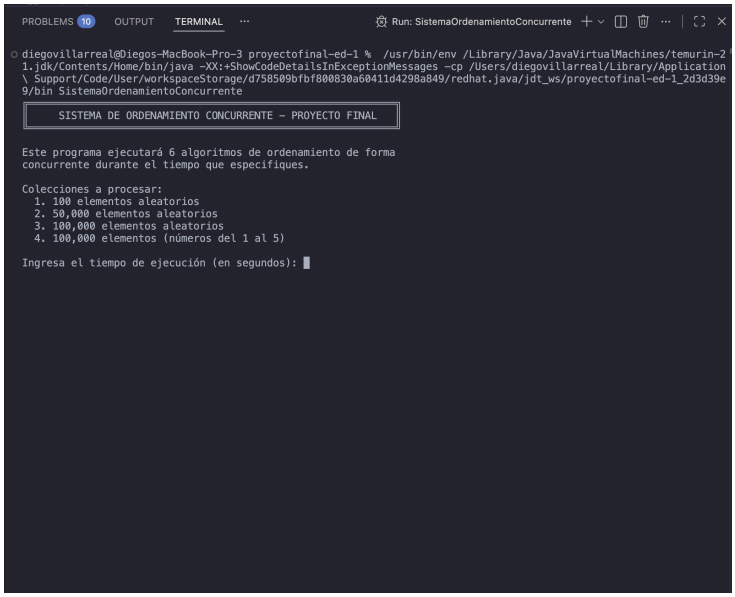
Interacción del usuario

Archivo: SistemaOrdenamientoConcurrente.java
El programa simplemente solicita el tiempo de ejecucion

```java

    private static int leerTiempoDelUsuario() {
        System.out.print(s: "Ingresa el tiempo de ejecución (en segundos): ");

        try {
            StringBuilder entrada = new StringBuilder();
            int caracter;

            while ((caracter = System.in.read()) != '\n' && caracter != -1) {
                if (caracter != '\r') {
                    entrada.append((char) caracter);
                }
            }

            String texto = entrada.toString().trim();
            return convertirAEntero(texto);

        } catch (Exception e) {
            System.out.println("Error al leer entrada: " + e.getMessage());
            return 30;
        }
    }
```

Si no se introduce nada / caracter invalido, se puede “catchear” con el catch

Terminal con la ejecucion:



Ejecucion concurrente

Archivo: HiloOrdenamiento.java
Cada algoritmo se ejecuto en su propio hilo:

```
You, 53 minutes ago | 1 author (You)
1  public class HiloOrdenamiento extends Thread {      You, 52 minutes ago • added readme, ad
2
3      private final TareaOrdenamiento.TipoAlgoritmo algoritmo;
4      private final GeneradorColecciones generador;
5      private final EstadisticasAlgoritmo estadisticas;
6      private final long tiempoLimiteMillis;
7      private final int[] tiposColecciones;
8      private volatile boolean debeDetenerse;
9
10     public HiloOrdenamiento(TareaOrdenamiento.TipoAlgoritmo algoritmo,
11                             long tiempoLimiteMillis,
12                             int[] tiposColecciones) {
13         this.algoritmo = algoritmo;
14         this.tiempoLimiteMillis = tiempoLimiteMillis;
15         this.tiposColecciones = tiposColecciones;
16         this.generador = new GeneradorColecciones();
17         this.estadisticas = new EstadisticasAlgoritmo(algoritmo.toString());
18         this.debeDetenerse = false;
19     }
```

Su ciclo de ejecucion:

```
25
26         while (!debeDetenerse) {
27             long tiempoActual = System.currentTimeMillis();
28             long tiempoTranscurrido = tiempoActual - tiempoInicio;
29
30             if (tiempoTranscurrido >= tiempoLimiteMillis) {
31                 break;
32             }
```

Contar colecciones ordenadas

Archivo: EstadisticasAlgoritmo.java
Cada algoritmo registra:

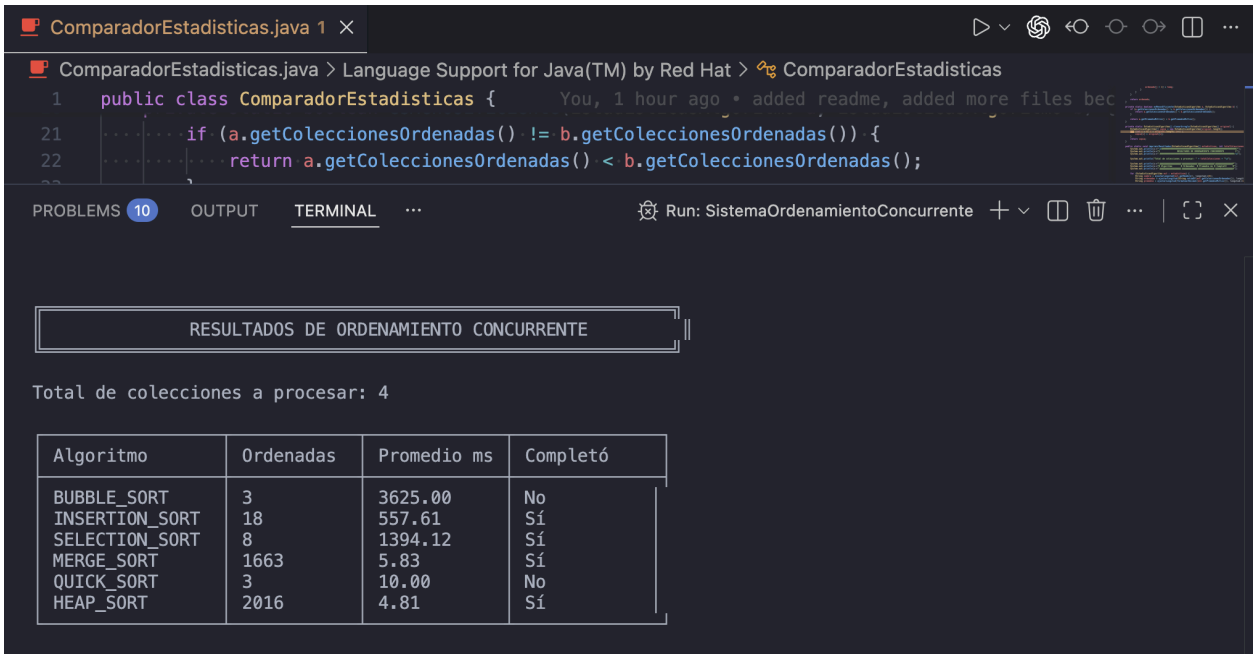
```
You, 59 minutes ago | 1 author (You)
public class EstadisticasAlgoritmo {      You, 57 minutes ago • added readme, added more file
    private final String nombre;
    private int coleccionesOrdenadas;
    private long tiempoTotalMillis;
    private boolean completoTodas;

    public EstadisticasAlgoritmo(String nombre) {
        this.nombre = nombre;
        this.coleccionesOrdenadas = 0;
        this.tiempoTotalMillis = 0;
        this.completoTodas = false;
    }
```

Este archivo es “Thread-Safe”
Usa **synchronized** para que los multiples hilos no corrompan los datos, y solo 1 hilo se puede actualizar a la vez

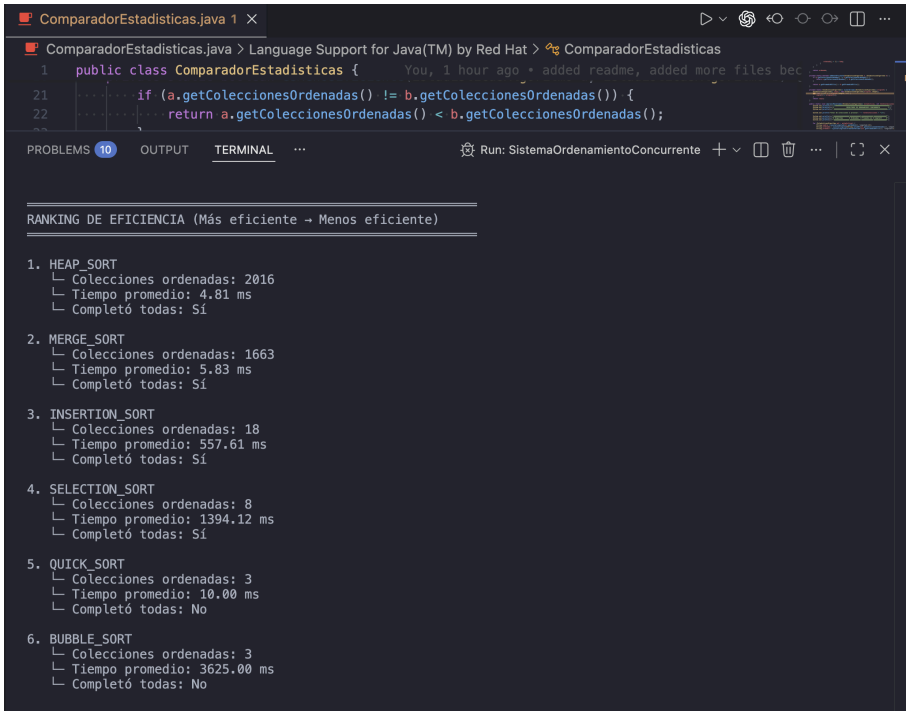
```
    public synchronized void registrarOrdenamiento(long tiempoMillis) {
        coleccionesOrdenadas++;
        tiempoTotalMillis += tiempoMillis;
    }

    public synchronized void marcarComoCompleto() {
        completoTodas = true;
    }

    public String getNombre() {
        return nombre;
    }

    public int getColeccionesOrdenadas() {
        return coleccionesOrdenadas;
    }

    public long getTiempoTotalMillis() {
        return tiempoTotalMillis;
    }

    public double getPromedioMillis() {
        if (coleccionesOrdenadas == 0) {
            return 0.0;
        }
        return (double) tiempoTotalMillis / coleccionesOrdenadas;
    }

    public boolean completoTodas() {
        return completoTodas;
    }
}
```

Salida del programa

Archivo: ComparadorEstadisticas.java
Este archivo genera 3 tipos de reportes: CON EJEMPLOS DE NUESTRO CODIGO (10 sec)
1.Informa cuantas colecciones ordeno cada algoritmo



2.Ranking de eficiencia (mas eficiente -> menos eficiente)



3.Algoritmos que completaron todas las colecciones

```
═══════════════════════════════════════════════════════════
ALGORITMOS QUE COMPLETARON TODAS LAS COLECCIONES
═══════════════════════════════════════════════════════════

✓ INSERTION_SORT
✓ SELECTION_SORT
✓ MERGE_SORT
✓ HEAP_SORT
```

4.Reporte tecnico

```
3. RESUMEN COMPARATIVO
──────────────────────────────────────────────────────────
```

Algoritmo	Mejor	Promedio	Peor
Bubble Sort	O(n)	O(n²)	O(n²)
Insertion Sort	O(n)	O(n²)	O(n²)
Selection Sort	O(n²)	O(n²)	O(n²)
Merge Sort	O(n logn)	O(n log n)	O(n logn)
Quick Sort	O(n logn)	O(n log n)	O(n²)
Heap Sort	O(n logn)	O(n log n)	O(n logn)

```
4. RECOMENDACIONES DE USO
──────────────────────────────────────────────────────────
• Datos pequeños o casi ordenados: Insertion Sort
• Datos grandes, garantía de rendimiento: Merge Sort o Heap Sort
• Mejor rendimiento promedio: Quick Sort
• Mínima memoria adicional: Heap Sort o Quick Sort
• Estabilidad requerida: Merge Sort o Bubble Sort


╔═══════════════════════════════════════════════════════╗
║                  EJECUCIÓN FINALIZADA                 ║
╚═══════════════════════════════════════════════════════╝
diegovillarreal@Diegos-MacBook-Pro-3 proyectofinal-ed-1 %
```

Uso de hilos (threads)

Archivo: HiloOrdenamiento.java

```java
public class HiloOrdenamiento extends Thread {
    @Override
    public void run() {
        // Código que se ejecuta en el hilo
    }
}
```

En este caso, cada algoritmo tiene su propio hilo independiente

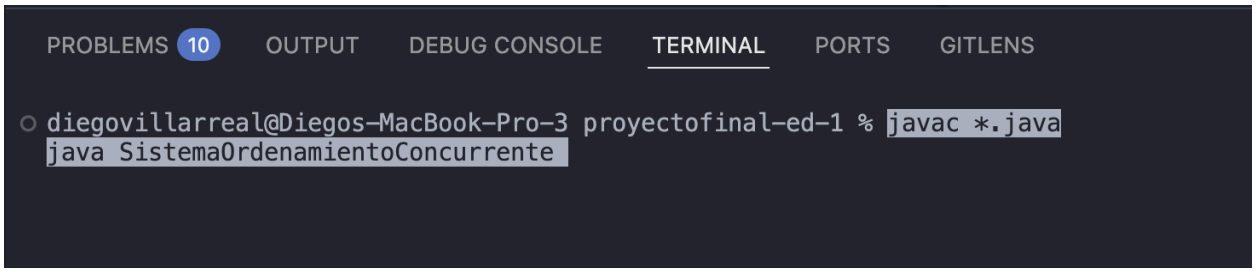
Generación de colecciones

Como no se implementó el “java.util”, optamos por hacer otro approach

```
    private int[] generarAleatorios(int cantidad, int valorMaximo) {
        int[] datos = new int[cantidad];

        for (int i = 0; i < cantidad; i++) {
            semilla = (semilla * 1103515245 + 12345) & 0x7fffffffL;
            datos[i] = ((int)(semilla % valorMaximo)) + 1;
        }

        return datos;
    }
```

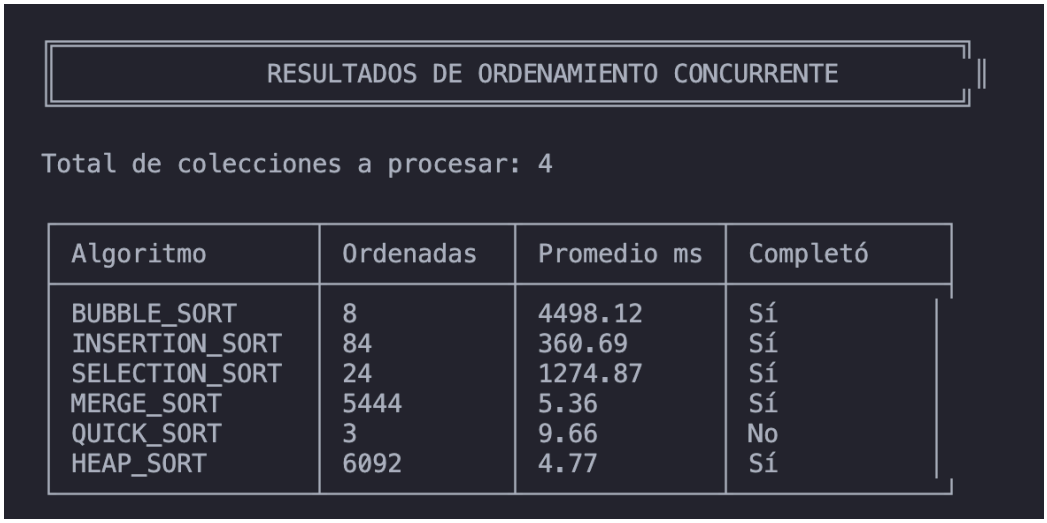
Ejecución del código

Primero se tienen que escribir dos comandos



Después se escribe el tiempo de ejecucion que se desea (en segundos)

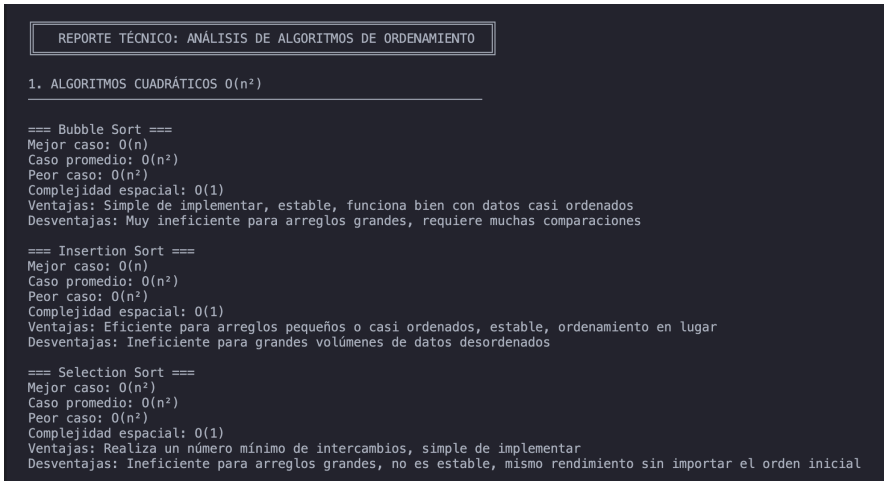
En este caso se usara 30 segundos, y se podra ver como unos algoritmos “compiten” para ver cual es mas rapido



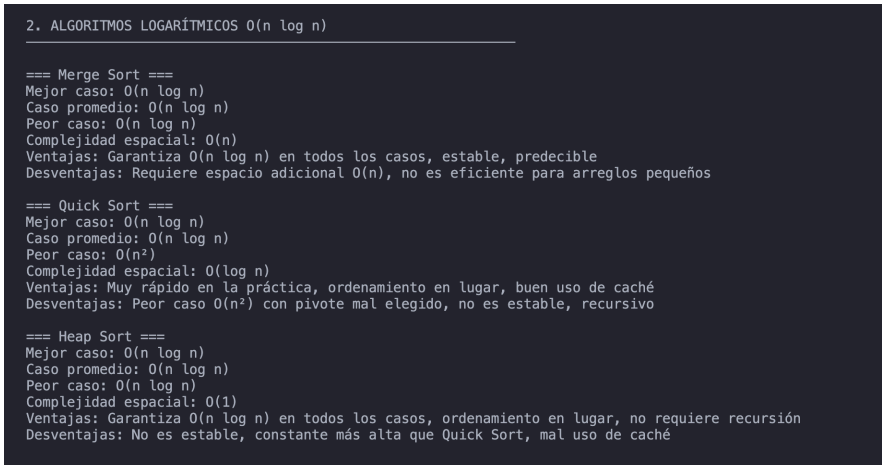
Aqui se vera el ranking de eficiencia, se puede observar donde Heap fue el mejor, y quick el peor, sin si quiera terminar todas las colecciones

```
═══════════════════════════════════════════════════
RANKING DE EFICIENCIA (Más eficiente → Menos eficiente)
═══════════════════════════════════════════════════

1. HEAP_SORT
   └─ Colecciones ordenadas: 6092
   └─ Tiempo promedio: 4.77 ms
   └─ Completó todas: Sí

2. MERGE_SORT
   └─ Colecciones ordenadas: 5444
   └─ Tiempo promedio: 5.36 ms
   └─ Completó todas: Sí

3. INSERTION_SORT
   └─ Colecciones ordenadas: 84
   └─ Tiempo promedio: 360.69 ms
   └─ Completó todas: Sí

4. SELECTION_SORT
   └─ Colecciones ordenadas: 24
   └─ Tiempo promedio: 1274.87 ms
   └─ Completó todas: Sí

5. BUBBLE_SORT
   └─ Colecciones ordenadas: 8
   └─ Tiempo promedio: 4498.12 ms
   └─ Completó todas: Sí

6. QUICK_SORT
   └─ Colecciones ordenadas: 3
   └─ Tiempo promedio: 9.66 ms
   └─ Completó todas: No
```

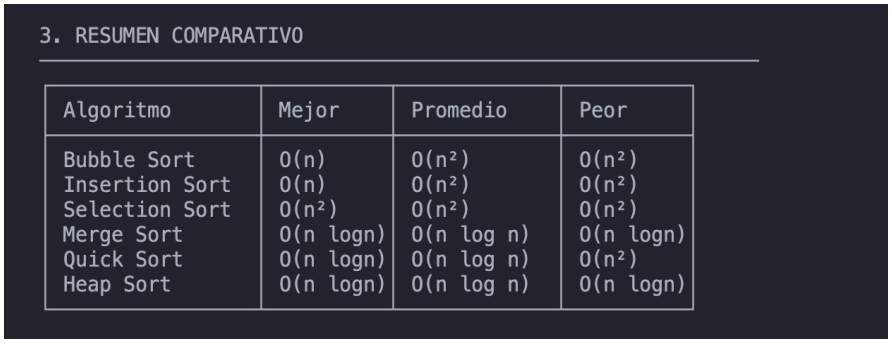
El reporte tecnico, aqui son cosas un poco mas complejas que se explica



```
REPORTE TÉCNICO: ANÁLISIS DE ALGORITMOS DE ORDENAMIENTO

1. ALGORITMOS CUADRÁTICOS O(n²)

=== Bubble Sort ===
Mejor caso: O(n)
Caso promedio: O(n²)
Peor caso: O(n²)
Complejidad espacial: O(1)
Ventajas: Simple de implementar, estable, funciona bien con datos casi ordenados
Desventajas: Muy ineficiente para arreglos grandes, requiere muchas comparaciones

=== Insertion Sort ===
Mejor caso: O(n)
Caso promedio: O(n²)
Peor caso: O(n²)
Complejidad espacial: O(1)
Ventajas: Eficiente para arreglos pequeños o casi ordenados, estable, ordenamiento en lugar
Desventajas: Ineficiente para grandes volúmenes de datos desordenados

=== Selection Sort ===
Mejor caso: O(n²)
Caso promedio: O(n²)
Peor caso: O(n²)
Complejidad espacial: O(1)
Ventajas: Realiza un número mínimo de intercambios, simple de implementar
Desventajas: Ineficiente para arreglos grandes, no es estable, mismo rendimiento sin importar el orden inicial
```

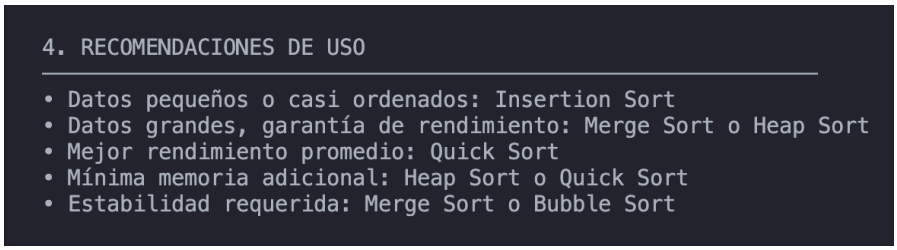


```
2. ALGORITMOS LOGARÍTMICOS O(n log n)

=== Merge Sort ===
Mejor caso: O(n log n)
Caso promedio: O(n log n)
Peor caso: O(n log n)
Complejidad espacial: O(n)
Ventajas: Garantiza O(n log n) en todos los casos, estable, predecible
Desventajas: Requiere espacio adicional O(n), no es eficiente para arreglos pequeños

=== Quick Sort ===
Mejor caso: O(n log n)
Caso promedio: O(n log n)
Peor caso: O(n²)
Complejidad espacial: O(log n)
Ventajas: Muy rápido en la práctica, ordenamiento en lugar, buen uso de caché
Desventajas: Peor caso O(n²) con pivote mal elegido, no es estable, recursivo

=== Heap Sort ===
Mejor caso: O(n log n)
Caso promedio: O(n log n)
Peor caso: O(n log n)
Complejidad espacial: O(1)
Ventajas: Garantiza O(n log n) en todos los casos, ordenamiento en lugar, no requiere recursión
Desventajas: No es estable, constante más alta que Quick Sort, mal uso de caché
```



3. RESUMEN COMPARATIVO

Algoritmo	Mejor	Promedio	Peor
Bubble Sort	O(n)	O(n²)	O(n²)
Insertion Sort	O(n)	O(n²)	O(n²)
Selection Sort	O(n²)	O(n²)	O(n²)
Merge Sort	O(n logn)	O(n log n)	O(n logn)
Quick Sort	O(n logn)	O(n log n)	O(n²)
Heap Sort	O(n logn)	O(n log n)	O(n logn)

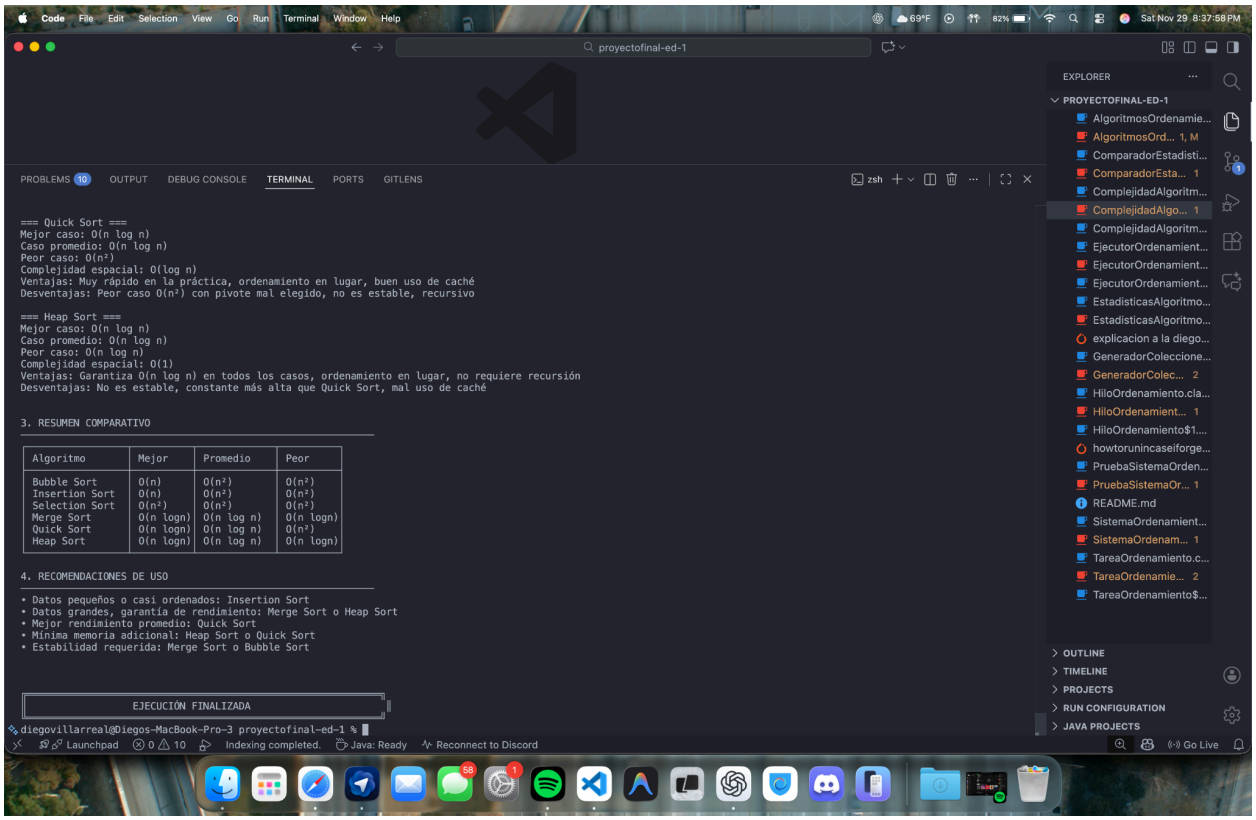


4. RECOMENDACIONES DE USO

- Datos pequeños o casi ordenados: Insertion Sort
- Datos grandes, garantía de rendimiento: Merge Sort o Heap Sort
- Mejor rendimiento promedio: Quick Sort
- Mínima memoria adicional: Heap Sort o Quick Sort
- Estabilidad requerida: Merge Sort o Bubble Sort

Final

```
EJECUCIÓN FINALIZADA
diegovillarreal@Diegos-MacBook-Pro-3 proyectofinal-ed-1 %
```



Conclusiones:

Ricardo Delgado: Con este proyecto puedo decir que me quedó mucho más claro cómo funcionan realmente los algoritmos de ordenamiento porque pude verlos trabajar al mismo tiempo. También fue interesante ver la carrera entre ellos y notar cómo uno termina muchísimo más rápido que otro.

También, el tener que programar cosas manuales sin usar ayudas de Java (como el `java.util`) fue un reto, pero me sirvió para entender qué es lo que pasa detrás del código. Al final comprendí que saber escoger el método correcto es lo más importante para que el programa no se tarde demasiado cuando son muchos datos.

Diego Villarreal: Este proyecto demuestra cómo diferentes algoritmos de ordenamiento trabajan al mismo tiempo comparando su velocidad y eficiencia. Al ejecutar los seis métodos simultáneamente durante un tiempo específico, podemos ver claramente cuáles son más rápidos en situaciones reales. Los algoritmos avanzados como Quick Sort y Merge Sort ordenan muchas más colecciones que los básicos como Bubble Sort. Esto nos enseña que la elección del algoritmo correcto es importante cuando trabajamos con grandes cantidades de datos. El sistema mide automáticamente todo y presenta resultados fáciles de entender, mostrando que la programación concurrente permite hacer varias tareas eficientemente.