Universidad de Cundinamarca

Estructura de Datos

R3-A2-S14 Video participativo

Javier Mateo Barrero Vanegas

Jim Alejandro Quiñones Martínez

Luis Ángel Martínez Cuenca

Harick Yesid Villarraga Rincon

Cristian Esteban Ruiz Parra

Docente: Dilia Ines Molina Cubillos

1 de noviembre de 2025

1. Introducción

El ordenamiento de datos es una operación fundamental en la ciencia de la computación, siendo esencial para mejorar la eficiencia de otros algoritmos como la búsqueda. Un algoritmo de ordenamiento tiene como propósito reorganizar los elementos de una colección (como un arreglo o lista) en una secuencia específica, ya sea ascendente o descendente.

Este informe tiene como objetivo principal presentar y comparar los algoritmos de ordenamiento más comunes, analizando su funcionamiento, ventajas y, críticamente, su eficiencia temporal, medida a través de la notación de la Gran O. Entender estas diferencias es crucial para la selección del método más adecuado en diversas aplicaciones informáticas.

2. Objetivos

2.1. Objetivo General

Analizar los principios de funcionamiento y comparar la eficiencia (complejidad temporal) de los algoritmos de ordenamiento fundamentales para determinar su aplicabilidad en diferentes volúmenes de datos.

2.2. Objetivos Específicos

- Describir el mecanismo de los algoritmos de ordenamiento simples (Burbuja, Selección, Inserción) e identificar las razones de su ineficiencia para grandes conjuntos de datos.
- Explicar la técnica "Divide y Vencerás" aplicada a los algoritmos eficientes (Mezcla, Rápido, Montículo) y su impacto en la complejidad temporal.
- Sintetizar las características de estabilidad y el uso de espacio auxiliar de cada método.

3. Marco Teórico

```
4. // Importamos la clase Scanner para leer la entrada del usuario
5. import java.util.Scanner;
6.
7. /**
8.  * Clase que implementa diferentes metodos de ordenamiento de
   arreglos
9.  * Incluye: Burbuja, Seleccion y Quicksort
10. */
11.public class MetodosOrdenamiento {
12.
13.    /**
```

```
14.      * Metodo de ordenamiento Burbuja (Bubble Sort)
15.      * Compara elementos adyacentes y los intercambia si estan en
   orden incorrecto
16.      * Complejidad temporal: O(n²)
17.      * @param arreglo - El arreglo de enteros a ordenar
18.      */
19.     public static void burbuja(int[] arreglo) {
20.         int n = arreglo.length; // Obtenemos el tamanio del arreglo
21.
22.         // Bucle externo: controla el numero de pasadas
23.         for (int i = 0; i < n - 1; i++) {
24.             // Bucle interno: compara elementos adyacentes
25.             // Con cada pasada, el elemento mas grande "burbujea"
   hacia el final
26.             for (int j = 0; j < n - i - 1; j++) {
27.                 // Si el elemento actual es mayor que el siguiente,
   intercambiarlos
28.                 if (arreglo[j] > arreglo[j + 1]) {
29.                     // Intercambio usando una variable temporal
30.                     int temp = arreglo[j];
31.                     arreglo[j] = arreglo[j + 1];
32.                     arreglo[j + 1] = temp;
33.                 }
34.             }
35.         }
36.     }
37.
38.     /**
39.      * Metodo de ordenamiento por Seleccion (Selection Sort)
40.      * Busca el elemento minimo y lo coloca en su posicion correcta
41.      * Complejidad temporal: O(n²)
42.      * @param arreglo - El arreglo de enteros a ordenar
43.      */
44.     public static void seleccion(int[] arreglo) {
45.         int n = arreglo.length; // Obtenemos el tamanio del arreglo
46.
47.         // Recorremos todo el arreglo menos el ultimo elemento
48.         for (int i = 0; i < n - 1; i++) {
49.             // Asumimos que el elemento actual es el minimo
50.             int minIndex = i;
51.
52.             // Buscamos el elemento minimo en el resto del arreglo
53.             for (int j = i + 1; j < n; j++) {
54.                 // Si encontramos un elemento menor, actualizamos
   minIndex
```

```
55.                 if (arreglo[j] < arreglo[minIndex]) {
56.                     minIndex = j;
57.                 }
58.             }
59.
60.             // Intercambiamos el minimo encontrado con el elemento
    en la posicion i
61.             int temp = arreglo[i];
62.             arreglo[i] = arreglo[minIndex];
63.             arreglo[minIndex] = temp;
64.         }
65.     }
66.
67.     /**
68.      * Metodo de ordenamiento Quicksort
69.      * Algoritmo recursivo de divide y venceras que usa un pivote
70.      * Complejidad temporal promedio: O(n log n)
71.      * @param arreglo - El arreglo de enteros a ordenar
72.      * @param inicio - Indice inicial del segmento a ordenar
73.      * @param fin - Indice final del segmento a ordenar
74.      */
75.     public static void quicksort(int[] arreglo, int inicio, int fin)
    {
76.         // Caso base: si hay mas de un elemento en el segmento
77.         if (inicio < fin) {
78.             // Particionar el arreglo y obtener el indice del pivote
79.             int pivoteIndex = particion(arreglo, inicio, fin);
80.
81.             // Ordenar recursivamente la parte izquierda del pivote
82.             quicksort(arreglo, inicio, pivoteIndex - 1);
83.
84.             // Ordenar recursivamente la parte derecha del pivote
85.             quicksort(arreglo, pivoteIndex + 1, fin);
86.         }
87.     }
88.
89.     /**
90.      * Metodo auxiliar para particionar el arreglo en Quicksort
91.      * Coloca el pivote en su posicion correcta y organiza los
    elementos
92.      * menores a la izquierda y mayores a la derecha
93.      * @param arreglo - El arreglo a particionar
94.      * @param inicio - Indice inicial del segmento
95.      * @param fin - Indice final del segmento
```

```
96.      * @return El indice donde quedo el pivote despues de la
   particion
97.      */
98.     private static int particion(int[] arreglo, int inicio, int fin)
   {
99.         // Elegimos el ultimo elemento como pivote
100.             int pivote = arreglo[fin];
101.
102.             // i es el indice del elemento mas pequenio
103.             int i = (inicio - 1);
104.
105.             // Recorremos desde inicio hasta fin-1
106.             for (int j = inicio; j < fin; j++) {
107.                 // Si el elemento actual es menor o igual al
   pivote
108.                 if (arreglo[j] <= pivote) {
109.                     i++; // Incrementamos el indice del elemento
   mas pequenio
110.
111.                     // Intercambiamos arreglo[i] con arreglo[j]
112.                     int temp = arreglo[i];
113.                     arreglo[i] = arreglo[j];
114.                     arreglo[j] = temp;
115.                 }
116.             }
117.
118.             // Colocamos el pivote en su posicion correcta
119.             int temp = arreglo[i + 1];
120.             arreglo[i + 1] = arreglo[fin];
121.             arreglo[fin] = temp;
122.
123.             // Retornamos la posicion del pivote
124.             return i + 1;
125.         }
126.
127.         /**
128.          * Metodo para mostrar el contenido del arreglo en consola
129.          * @param arreglo - El arreglo a mostrar
130.          */
131.         public static void mostrarArreglo(int[] arreglo) {
132.             // Recorremos cada elemento del arreglo
133.             for (int num : arreglo) {
134.                 System.out.print(num + " "); // Imprimimos el
   numero seguido de un espacio
135.             }
```

```
136.             System.out.println(); // Salto de linea al final
137.         }
138.
139.         /**
140.          * Metodo para solicitar al usuario que ingrese los datos
   del arreglo
141.          * @param sc - El objeto Scanner para leer la entrada
142.          * @return El arreglo con los datos ingresados por el
   usuario
143.          */
144.         public static int[] ingresarDatos(Scanner sc) {
145.             System.out.print("\nIngrese la cantidad de elementos
   del arreglo: ");
146.             int n = sc.nextInt();
147.
148.             // Creamos el arreglo con el tamanio especificado
149.             int[] arreglo = new int[n];
150.
151.             // Solicitamos cada elemento al usuario
152.             System.out.println("Ingrese los " + n + "
   elementos:");
153.             for (int i = 0; i < n; i++) {
154.                 System.out.print("Elemento " + (i + 1) + ": ");
155.                 arreglo[i] = sc.nextInt();
156.             }
157.
158.             return arreglo;
159.         }
160.
161.         /**
162.          * Metodo principal - punto de entrada del programa
163.          * Muestra un menu interactivo para elegir el metodo de
   ordenamiento
164.          */
165.         public static void main(String[] args) {
166.             // Creamos un objeto Scanner para leer la entrada del
   usuario
167.             Scanner sc = new Scanner(System.in);
168.
169.             int opcion; // Variable para almacenar la opcion del
   menu
170.             int[] arreglo = null; // Arreglo que contendra los
   datos del usuario
171.
```

```
172.             // Bucle do-while para mostrar el menu hasta que el
   usuario elija salir
173.             do {
174.                 // Mostramos el menu de opciones
175.                 System.out.println("\n--- MENU DE ORDENAMIENTO ---
   ");
176.                 System.out.println("1. Ingresar nuevos datos");
177.                 System.out.println("2. Ordenar con Metodo
   Burbuja");
178.                 System.out.println("3. Ordenar con Metodo
   Seleccion");
179.                 System.out.println("4. Ordenar con Metodo
   Quicksort");
180.                 System.out.println("5. Salir");
181.                 System.out.print("Seleccione una opcion: ");
182.
183.                 // Leemos la opcion del usuario
184.                 opcion = sc.nextInt();
185.
186.                 // Evaluamos la opcion seleccionada usando switch
187.                 switch (opcion) {
188.                     case 1 -> { // Opcion 1: Ingresar datos
189.                         arreglo = ingresarDatos(sc);
190.                         System.out.println("\nDatos ingresados
   correctamente:");
191.                         mostrarArreglo(arreglo);
192.                     }
193.                     case 2 -> { // Opcion 2: Metodo Burbuja
194.                         if (arreglo == null) {
195.                             System.out.println("\nError: Primero
   debe ingresar los datos (opcion 1)");
196.                         } else {
197.                             int[] copia = arreglo.clone(); //
   Creamos una copia para no modificar el original
198.                             System.out.println("\nArreglo
   original:");
199.                             mostrarArreglo(copia);
200.                             burbuja(copia); // Ordenamos con
   burbuja
201.                             System.out.println("Arreglo ordenado
   (Burbuja):");
202.                             mostrarArreglo(copia);
203.                         }
204.                     }
205.                     case 3 -> { // Opcion 3: Metodo Seleccion
```

```
206.                         if (arreglo == null) {
207.                             System.out.println("\nError: Primero
    debe ingresar los datos (opcion 1)");
208.                         } else {
209.                             int[] copia = arreglo.clone(); //
    Creamos una copia para no modificar el original
210.                             System.out.println("\nArreglo
    original:");
211.                             mostrarArreglo(copia);
212.                             seleccion(copia); // Ordenamos con
    seleccion
213.                             System.out.println("Arreglo ordenado
    (Seleccion):");
214.                             mostrarArreglo(copia);
215.                         }
216.                     }
217.                     case 4 -> { // Opcion 4: Metodo Quicksort
218.                         if (arreglo == null) {
219.                             System.out.println("\nError: Primero
    debe ingresar los datos (opcion 1)");
220.                         } else {
221.                             int[] copia = arreglo.clone(); //
    Creamos una copia para no modificar el original
222.                             System.out.println("\nArreglo
    original:");
223.                             mostrarArreglo(copia);
224.                             quicksort(copia, 0, copia.length - 1);
    // Ordenamos con quicksort
225.                             System.out.println("Arreglo ordenado
    (Quicksort):");
226.                             mostrarArreglo(copia);
227.                         }
228.                     }
229.                     case 5 -> System.out.println("\nSaliendo del
    programa..."); // Opcion 5: Salir
230.                     default -> System.out.println("\nOpcion
    invalida. Intente nuevamente."); // Opcion no valida
231.                 }
232.             } while (opcion != 5); // Continuamos hasta que el
    usuario elija la opcion 5
233.
234.             // Cerramos el Scanner para liberar recursos
235.             sc.close();
236.         }
237.     }
```

238. Métodos de Ordenamiento

Los métodos de ordenamiento son algoritmos diseñados para reorganizar elementos (como números, textos u objetos) en un orden específico (ascendente o descendente). Son fundamentales en informática porque optimizan operaciones como búsquedas, análisis de datos y gestión de información.

238.1. Algoritmos Simples (Ineficientes para Grandes Datos)

Estos algoritmos son fáciles de entender e implementar, pero su rendimiento es $O(N^2)$, por lo que son ineficientes para grandes conjuntos de datos.

238.1.1. Ordenamiento de Burbuja (Bubble Sort)

Concepto: Compara elementos adyacentes y los intercambia si están en desorden, repitiendo el proceso hasta que no se necesiten más intercambios.

Complejidad: $O(N^2)$ en el peor y caso promedio. $O(N)$ en el mejor caso.

Estable: Sí.

Uso: Educativo, no recomendado para datos grandes.

238.1.2. Ordenamiento por Selección (Selection Sort)

Concepto: Busca el elemento mínimo/máximo en la lista desordenada y lo coloca en su posición correcta, repitiendo el proceso para el resto de la lista.

Complejidad: Siempre $O(N^2)$.

Estable: No.

Uso: Útil cuando el intercambio de elementos es costoso.

238.1.3. Ordenamiento por Inserción (Insertion Sort)

Concepto: Construye una sublista ordenada insertando cada elemento en su posición correcta dentro de la sublista ya ordenada.

Complejidad: $O(N^2)$ en el peor y caso promedio. $O(N)$ en el mejor caso.

Estable: Sí.

Uso: Eficiente para listas pequeñas o casi ordenadas.

238.2. Algoritmos Eficientes (Basados en "Divide y Vencerás")

Estos algoritmos logran una eficiencia de tiempo mucho mejor, típicamente $O(N \log N)$, mediante la división recursiva de la lista.

238.2.1. Ordenamiento por Mezcla (Merge Sort)

Concepto: Divide la lista en mitades, ordena cada mitad recursivamente y luego combina (mezcla) las mitades ordenadas.

Complejidad: Siempre $O(N \log N)$.

Estable: Sí.

Desventaja: Requiere espacio de memoria adicional ($O(N)$).

238.2.2. Ordenamiento Rápido (Quick Sort)

Concepto: Elige un "pivote", particiona la lista en elementos menores y mayores al pivote, y ordena recursivamente las particiones.

Complejidad: $O(N \log N)$ en el promedio. $O(N^2)$ en el peor caso.

Estable: No.

Ventaja: Generalmente más rápido en la práctica.

238.2.3. Ordenamiento por Montículo (Heap Sort)

Concepto: Construye un montículo (heap) máximo/mínimo a partir de los datos y extrae repetidamente el elemento raíz para colocarlo en su posición final.

Complejidad: Siempre $O(N \log N)$.

Estable: No.

Ventaja: No requiere memoria adicional significativa (in situ).

238.3. Algoritmos Especializados

Estos métodos pueden superar la barrera de $O(N \log N)$ al no depender de comparaciones, pero tienen requisitos estrictos sobre el tipo de datos (generalmente enteros en un rango limitado).

238.3.1. Counting Sort (Ordenamiento por Cuentas)

Concepto: Cuenta la frecuencia de cada elemento y calcula las posiciones finales directamente a partir de esas cuentas.

Complejidad: $O(N + K)$, donde K es el rango de valores.

Estable: Sí.

Requisito: Funciona solo con valores enteros no negativos en un rango limitado (K).

238.3.2. Radix Sort (Ordenamiento por Raíz)

Concepto: Ordena los números por dígitos o caracteres (unidad, decena, etc.), usando un algoritmo estable (como Counting Sort) en cada iteración.

Complejidad: $O(D \cdot N)$, donde D es el número de dígitos.

Estable: Sí.

Uso: Ideal para números o cadenas con longitud fija.

238.3.3. Bucket Sort (Ordenamiento por Cubetas)

Concepto: Distribuye los elementos en "cubetas" (o buckets) según su valor, ordena cada cubeta (usando otro algoritmo) y concatena los resultados.

Complejidad: Promedio $O(N + K)$, peor caso $O(N^2)$.

Requisito: Datos distribuidos uniformemente.

239. Comparación General y Complejidad Temporal

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso	Estable	Memoria Extra
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Sí	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Sí	$O(1)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Sí	$O(N)$

Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	No	$O(\log N)$
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	No	$O(1)$
Counting Sort	$O(N + K)$	$O(N + K)$	$O(N + K)$	Sí	$O(K)$
Radix Sort	$O(D \cdot N)$	$O(D \cdot N)$	$O(D \cdot N)$	Sí	$O(N + D)$

240. Guía de Selección y Ejemplo

240.1. Criterios de Selección

La elección del algoritmo depende de los requisitos del sistema:

- Listas Pequeñas o Casi Ordenadas: Insertion Sort.
- Listas Grandes, General: Quick Sort (más rápido en promedio).
- Cuando la Estabilidad es Crítica: Merge Sort.
- Recursos de Memoria Limitados: Heap Sort (requiere $O(1)$ de espacio auxiliar).
- Datos Enteros en Rango Fijo: Counting Sort o Radix Sort.

240.2. Ejemplo Visual (Quick Sort)

Pasos para ordenar [5, 2, 9, 1, 5]:

1. Elegir Pivote: Se elige un elemento como pivote (ejemplo: el último elemento, que es 5).
2. Particionar: Los elementos menores (2, 1) van a un lado, y los mayores (9) van al otro. La lista se convierte en [2, 1, 5, 9, 5].
3. Ordenar Recursivamente: Se ordenan las sublistas izquierda y derecha.
4. Combinar: El resultado final es la combinación: [1, 2, 5, 5, 9].

241. Conclusión

Los métodos de ordenamiento representan un equilibrio fundamental entre la simplicidad de implementación y la eficiencia de ejecución. Si bien los algoritmos cuadráticos ($O(N^2)$) como Bubble Sort son intuitivos, su uso está restringido a conjuntos de datos muy pequeños. Para colecciones de tamaño considerable, los algoritmos basados en "Divide y Vencerás" como Merge Sort y Quick Sort, con su complejidad $O(N \log N)$, son indispensables. La elección final entre ellos dependerá de factores como la necesidad de estabilidad (Merge Sort) o la limitación de espacio auxiliar (Quick Sort/Heap Sort *in situ*).