# Estructura de Datos Actividad 2

Carlos Esteban Lara Gil

Prof: John Arrieta



Cartagena de Indias 7 de Abril del 2024. Piedra Bolívar

Ingeniería de Software

## Tabla de contenido

[Estructura de Datos Actividad 2](#_cg8bvoeu8ww3)

[Tabla de contenido](#_l55comz9ww18)

[Introducción](#_cn1jaa106lph)

[Objetivos](#_n25y29qjwet1)

[Justificación](#_medklbhjzun9)

[Justificación del Análisis Comparativo](#_mobqnh8gt8ai)

[Desarrollo](#_9gq78wy2qqlg)

[Lenguajes de programación escogidos](#_5tg89yn40lqj)

[Algoritmos escogidos](#_nxokg9af1qod)

[Ordenamiento burbuja (Bubble sort).](#_pjrronv0ccm5)

[Ordenamiento aleatorio](#_9vck238jf3m8)

[Ordenamiento Rápido](#_quhdwge98kvd)

[Distribución Sort](#_qlr8nfbcpc6x)

[Ordenamiento combsort](#_azdz3b8g6dh4)

[Fuentes:](#_rp0582j2nzv2)

[Ordenamiento Burbuja](#_b35f5ww1h9tq)

[Ordenamiento aleatorio](#_4cwsj1vngqda)

[Ordenamiento rápido:](#_z7kiqpqvc7p8)

[Distribución Sort](#_jszdkiuvvm33)

[CombSort](#_6jfrw5vnac)

[Link de github: https://github.com/herculeskan/estructura\_datos\_act2](#_gmbk1qbv9cqn)

## Introducción

Las estructuras de datos son herramientas fundamentales en la informática para organizar y almacenar información de manera eficiente. En este ensayo, exploraremos algunas estructuras de datos comunes, junto con cinco algoritmos de ordenamiento: ordenamiento por burbuja, ordenamiento aleatorio, ordenamiento rápido, ordenamiento por distribución y ordenamiento combsort.

**Estructuras de Datos:**

* **Arreglos:** Una estructura simple que almacena una colección de elementos del mismo tipo, accesibles por un índice.
* **Listas enlazadas:** Una estructura dinámica que almacena elementos de forma no contigua, enlazando cada elemento con el siguiente.
* **Árboles binarios de búsqueda:** Una estructura jerárquica que ordena elementos de forma eficiente, permitiendo búsquedas y operaciones eficientes.

**Algoritmos de Ordenamiento:**

* **Ordenamiento por burbuja:** Un algoritmo simple que compara elementos adyacentes y los intercambia si están en desorden. Es fácil de entender pero tiene una complejidad temporal de O(n^2).
* **Ordenamiento aleatorio:** Una variante del ordenamiento rápido que introduce aleatoriedad en la selección del pivote. Puede mejorar el rendimiento en algunos casos, pero su eficiencia depende de la distribución de los datos.
* **Ordenamiento rápido:** Un algoritmo eficiente que utiliza la técnica de "dividir y vencerás" para ordenar elementos. Tiene una complejidad temporal promedio de O(n log n), pero puede ser ineficiente en el peor caso.
* **Ordenamiento por distribución:** Un algoritmo que aprovecha la distribución de valores en el conjunto de datos para ordenarlo de forma eficiente. Es ideal para conjuntos con un rango de valores limitado.
* **Ordenamiento combsort:** Un algoritmo que combina características del ordenamiento por burbuja y el ordenamiento por salto. Es simple de implementar y eficiente en promedio, pero no tan estable como otros algoritmos.

**En este análisis:**

* Analizaremos las ventajas y desventajas de cada estructura de datos y algoritmo de ordenamiento.
* Exploraremos la complejidad temporal y espacial de cada algoritmo.
* Discutiremos la elección adecuada de la estructura de datos y algoritmo de ordenamiento para diferentes casos de uso.
* Mostraremos ejemplos de implementación de cada algoritmo en Python.

Al comprender las estructuras de datos y los algoritmos de ordenamiento, podemos elegir las herramientas adecuadas para organizar y manipular información de forma eficiente, optimizando el rendimiento de nuestros programas.

## Objetivos

**Objetivo principal:**

* **Analizar y comparar cinco algoritmos de ordenamiento:** ordenamiento por burbuja, ordenamiento aleatorio, ordenamiento rápido, ordenamiento por distribución y ordenamiento combsort, en los lenguajes Java, JavaScript y Python.

**Objetivos específicos:**

* **Describir las características de cada algoritmo de ordenamiento:**
  + Funcionamiento
  + Eficiencia (complejidad temporal y espacial)
  + Ventajas y desventajas
  + Estabilidad
* **Implementar cada algoritmo de ordenamiento en los tres lenguajes.**
* **Comparar el rendimiento de los algoritmos de ordenamiento en diferentes casos de uso en cada lenguaje.**
* **Discutir la elección adecuada del algoritmo de ordenamiento para diferentes situaciones en cada lenguaje.**
* **Explorar las estructuras de datos utilizadas en los algoritmos de ordenamiento.**
* **Analizar la importancia de las estructuras de datos y algoritmos de ordenamiento en la informática.**

**Resultados esperados:**

* **Una descripción completa de los cinco algoritmos de ordenamiento en los tres lenguajes.**
* **Implementaciones funcionales de los algoritmos en Java, JavaScript y Python.**
* **Un análisis comparativo del rendimiento de los algoritmos en cada lenguaje.**
* **Recomendaciones para elegir el algoritmo adecuado en diferentes situaciones y lenguajes.**
* **Una mayor comprensión de la importancia de las estructuras de datos y algoritmos de ordenamiento en la informática.**

**Este análisis será de utilidad para:**

* **Estudiantes de informática:** Para comprender mejor los algoritmos de ordenamiento y las estructuras de datos.
* **Programadores:** Para elegir el algoritmo de ordenamiento adecuado para sus proyectos en diferentes lenguajes.
* **Cualquier persona interesada en la informática:** Para aprender sobre las herramientas fundamentales para organizar y manipular información.

**Esperamos que este análisis sea una valiosa contribución al conocimiento sobre las estructuras de datos y los algoritmos de ordenamiento en Java, JavaScript y Python.**

**Metodología:**

* **Revisión de la literatura:** Se analizarán libros, artículos y recursos web sobre estructuras de datos y algoritmos de ordenamiento.
* **Implementación de los algoritmos:** Se implementarán los cinco algoritmos de ordenamiento en Java, JavaScript y Python.
* **Experimentación:** Se realizarán pruebas para comparar el rendimiento de los algoritmos en diferentes casos de uso en cada lenguaje.
* **Análisis de resultados:** Se analizarán los resultados de las pruebas y se discutirán las conclusiones.

## Justificación

## **Justificación del Análisis Comparativo**

**Importancia de las Estructuras de Datos y Algoritmos de Ordenamiento:**

* Las estructuras de datos son esenciales para organizar y almacenar información de forma eficiente.
* Los algoritmos de ordenamiento permiten ordenar y manipular datos de forma eficiente.
* La elección adecuada de la estructura de datos y el algoritmo de ordenamiento puede mejorar significativamente el rendimiento de un programa.

**Necesidad de un Análisis Comparativo:**

* Existe una amplia variedad de algoritmos de ordenamiento con diferentes características.
* La elección del algoritmo adecuado depende de varios factores, como el tamaño del conjunto de datos, la distribución de los datos y el lenguaje de programación utilizado.
* No existe un único algoritmo de ordenamiento que sea el mejor en todas las situaciones.

**Beneficios del Análisis Comparativo:**

* Proporcionará una comprensión completa de cinco algoritmos de ordenamiento: ordenamiento por burbuja, ordenamiento aleatorio, ordenamiento rápido, ordenamiento por distribución y ordenamiento combsort.
* Comparará el rendimiento de los algoritmos en diferentes casos de uso en Java, JavaScript y Python.
* Ofrecerá recomendaciones para elegir el algoritmo adecuado en diferentes situaciones y lenguajes.
* Ayudará a los estudiantes de informática, programadores y cualquier persona interesada en la informática a comprender mejor las estructuras de datos y los algoritmos de ordenamiento.

**Contribución del Análisis Comparativo:**

* Será una valiosa referencia para la selección de algoritmos de ordenamiento en Java, JavaScript y Python.
* Ayudará a mejorar el rendimiento de los programas que utilizan estos algoritmos.
* Contribuirá a la comprensión de las estructuras de datos y su importancia en la informática.

## Desarrollo

### Lenguajes de programación escogidos

Con el fin de validar que el estudiante ha realizado un correcto y responsable su proceso de estudio, algunas preguntas deben ir acompañadas de un ejemplo simple escrito en al menos 3 de los siguientes lenguajes de programación estrictamente necesario que cada pregunta deba tener su respectivo

* Java
* Python
* JavaScript

### Algoritmos escogidos

Explicar al menos 5 de los siguientes algoritmos de ordenamiento. Debe usar un único conjunto de datos (array o lista) y escribir un ejemplo de cada algoritmo en al menos 3 lenguajes de programación de los descritos anteriormente. Se debe adjuntar a este informe (subiendo a la plataforma SIMA-PESAD) un archivo comprimido con el código de los ejemplos. Hoy dia no es necesario tener que instalar SDK o IDE en nuestro PC para poder desarrollar ejemplos simples de codigo, ya que existen muchas aplicaciones online y para el smartphone que nos permiten escribir el código, compilarlo, ejecutarlo y hasta depurarlo para hacer la respectiva prueba funcional.

* Ordenamiento de burbuja
* Ordenamiento aleatorio
* Ordenamiento rápido
* Distribución Sort
* Ordenamiento combsort

### Ordenamiento burbuja (Bubble sort).

El **ordenamiento burbuja** hace múltiples pasadas a lo largo de una lista. Compara los ítems adyacentes e intercambia los que no están en orden. Cada pasada a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado. En esencia, cada ítem “burbujea” hasta el lugar al que pertenece.

**Funcionamiento**:

1. Se comienza por comparar el primer elemento con el segundo. Si el primer elemento es mayor que el segundo, se intercambian sus posiciones.
2. Luego, se compara el segundo elemento con el tercero, y se intercambian si no están en orden.
3. Este proceso se repite hasta llegar al final de la lista
4. Se vuelve al principio de la lista y se repiten los pasos 1 a 3 hasta que no se realicen más intercambios. Esto significa que está ordenada.

**Ejemplo**:

Usemos la siguiente lista:

 [54,26,93,17,77,31,44,55,20]



# Ordenamiento burbuja con la lista [54,26,93,17,77,31,44,55,20]

def bubbleSort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

# mostrat el resultado en consola

arr = [54,26,93,17,77,31,44,55,20]

print("Lista original:\n ", arr)

print("\n----------- Ordenando la lista -----------\n")

print("\nLista ordenada por el metodo de burbuja: \n", bubbleSort(arr))



// bubble sort de la lista [54,26,93,17,77,31,44,55,20]

function bubbleSort(arr) {

let n = arr.length;

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

for (let j = 0; j < n-i-1; j++) {

if (arr[j] > arr[j+1]) {

let temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

return arr;

}

// mostrar el resultado en consola

arr=[54,26,93,17,77,31,44,55,20]

console.log("lista sin ordernar: "+arr); // [54, 26, 93, 17, 77, 31, 44, 55, 20]

console.log("Ordenamiento burbuja: "+bubbleSort(arr)); // [17, 20, 26, 31, 44, 54, 55, 77, 93]



package com.example;

//método para mostrar la lista

public class bubbleSort {

public static void mostrarLista(int[] lista) {

for (int i = 0; i < lista.length; i++) {

System.out.print(lista[i] + " ");

}

System.out.println();

}

public static void main(String[] args) {

// bubble sort de la liste [54,26,93,17,77,31,44,55,20]

int[] liste = { 54, 26, 93, 17, 77, 31, 44, 55, 20 };

System.out.println("Mostrar la lista sin ordenar");

mostrarLista(liste); // mostrar la liste sin ordenar

int n = liste.length;

int temp = 0;

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

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

if (liste[j - 1] > liste[j]) {

// echanges les elements

temp = liste[j - 1];

liste[j - 1] = liste[j];

liste[j] = temp;

}

}

}

// Mostrar lista ordenada

System.out.println("Mostrar la lista ordenada");

mostrarLista(liste);

}

}



**ventajas**:

* Es un algoritmo muy sencillo de entender e implementar
* No requiere memoria adicional.
* Es estable, lo que significa que conserva el orden original de los elementos iguales

**Desventajas**:

* Es un algoritmo ineficiente para ordenar grandes cantidades de datos
* Su complejidad temporal es O(n²), lo que significa que el tiempo de ejecución aumenta exponencialmente con el tamaño de la lista.

### **Ordenamiento aleatorio**

El ordenamiento aleatorio, también conocido como “suffiling” o “mezcla aleatoria”, es un proceso que reorganiza los elementos

El ordenamiento aleatorio, también conocido como "aleatorización", es una técnica utilizada en la informática para mejorar el rendimiento de algoritmos de ordenamiento en ciertas estructuras de datos. La idea es introducir un elemento de aleatoriedad en el proceso de ordenamiento, lo que puede ayudar a evitar situaciones en las que el algoritmo se atasca en un caso particular y no logra converger a una solución óptima.

**Estructuras de datos donde se aplica el ordenamiento aleatorio:**

* **Arreglos:** El ordenamiento aleatorio puede ser utilizado para ordenar arreglos de cualquier tipo de dato. Es especialmente útil para ordenar arreglos grandes, donde el ordenamiento tradicional puede ser demasiado lento.
* **Listas enlazadas:** El ordenamiento aleatorio también se puede utilizar para ordenar listas enlazadas. Sin embargo, en este caso, la aleatoriedad puede afectar la eficiencia del algoritmo, ya que el acceso a los elementos de la lista es secuencial.
* **Árboles binarios de búsqueda:** El ordenamiento aleatorio no se suele utilizar para ordenar árboles binarios de búsqueda, ya que estos árboles ya están ordenados por naturaleza.

**Algoritmos de ordenamiento aleatorio:**

* **Ordenamiento rápido aleatorio:** Este algoritmo es una variante del ordenamiento rápido tradicional que introduce un elemento de aleatoriedad en la selección del pivote. Esto ayuda a evitar que el algoritmo se atasque en casos particulares.
* **Ordenamiento por mezcla aleatoria:** Este algoritmo divide el problema en subproblemas más pequeños de forma aleatoria y luego los ordena de forma recursiva. La aleatoriedad ayuda a evitar que el algoritmo se atasque en casos particulares.

**Ventajas del ordenamiento aleatorio:**

* Puede mejorar el rendimiento de los algoritmos de ordenamiento en ciertos casos.
* Puede ayudar a evitar que los algoritmos de ordenamiento se atasquen en casos particulares.
* Es relativamente fácil de implementar.

**Desventajas del ordenamiento aleatorio:**

* Puede ser menos eficiente que el ordenamiento tradicional en algunos casos.
* Puede ser impredecible, ya que el resultado del ordenamiento depende de la aleatoriedad.
* Puede no ser adecuado para todas las estructuras de datos.

**Ejemplos**

package com.example;

public class OrdenamientoAleatorio {

// Ejemplo de ordenamiento aleatorio estructura de datos

public static void main(String[] args) {

int[] numeros = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int[] numerosAleatorios = new int[9];

int[] numerosOrdenados = new int[9];

int numeroAleatorio;

int contador = 0;

boolean existe = false;

// Generar números aleatorios

while (contador < 9) {

numeroAleatorio = (int) (Math.random() \* 9) + 1;

for (int i = 0; i < 9; i++) {

if (numerosAleatorios[i] == numeroAleatorio) {

existe = true;

break;

}

}

if (!existe) {

numerosAleatorios[contador] = numeroAleatorio;

contador++;

}

existe = false;

}

// Ordenar números aleatorios

for (int i = 0; i < 9; i++) {

numerosOrdenados[i] = numeros[numerosAleatorios[i] - 1];

}

// Mostrar números aleatorios

System.out.println("Números aleatorios:");

for (int i = 0; i < 9; i++) {

System.out.print(numerosAleatorios[i] + " ");

}

System.out.println();

// Mostrar números ordenados

System.out.println("Números ordenados:");

for (int i = 0; i < 9; i++) {

System.out.print(numerosOrdenados[i] + " ");

}

System.out.println();

}

}



// Ejemplo de ordenamiento aleatorio

let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9];

let numerosAleatorios = new Array(9).fill(0);

let numerosOrdenados = new Array(9).fill(0);

let numeroAleatorio;

let contador = 0;

let existe = false;

// Generar números aleatorios

while (contador < 9) {

numeroAleatorio = Math.floor(Math.random() \* 9) + 1;

for (let i = 0; i < 9; i++) {

if (numerosAleatorios[i] === numeroAleatorio) {

existe = true;

break;

}

}

if (!existe) {

numerosAleatorios[contador] = numeroAleatorio;

contador++;

}

existe = false;

}

// Ordenar números aleatorios

for (let i = 0; i < 9; i++) {

numerosOrdenados[i] = numeros[numerosAleatorios[i] - 1];

}

// Mostrar números aleatorios

console.log("Números aleatorios:");

for (let i = 0; i < 9; i++) {

process.stdout.write(numerosAleatorios[i] + " ");

}

console.log();

// Mostrar números ordenados

console.log("Números ordenados:");

for (let i = 0; i < 9; i++) {

process.stdout.write(numerosOrdenados[i] + " ");

}

console.log();



import random

# Ejemplo de ordenamiento aleatorio

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

numeros\_aleatorios = [0] \* 9

numeros\_ordenados = [0] \* 9

contador = 0

existe = False

# Generar números aleatorios

while contador < 9:

numero\_aleatorio = random.randint(1, 9)

for i in range(9):

if numeros\_aleatorios[i] == numero\_aleatorio:

existe = True

break

if not existe:

numeros\_aleatorios[contador] = numero\_aleatorio

contador += 1

existe = False

# Ordenar números aleatorios

for i in range(9):

numeros\_ordenados[i] = numeros[numeros\_aleatorios[i] - 1]

# Mostrar números aleatorios

print("Números aleatorios:")

for i in range(9):

print(numeros\_aleatorios[i], end=" ")

print()

# Mostrar números ordenados

print("Números ordenados:")

for i in range(9):

print(numeros\_ordenados[i], end=" ")

print()



### **Ordenamiento Rápido**

El ordenamiento rápido, también conocido como "quicksort", es un algoritmo de ordenamiento eficiente que utiliza la técnica de "dividir y vencerás" para ordenar elementos en una estructura de datos.

**Funcionamiento:**

1. **Elegir un pivote:** Se selecciona un elemento del conjunto de datos como pivote.
2. **Particionar:** Se divide el conjunto de datos en dos subconjuntos:
   * **Subconjunto izquierdo:** Contiene todos los elementos menores que el pivote.
   * **Subconjunto derecho:** Contiene todos los elementos mayores o iguales que el pivote.
3. **Ordenamiento recursivo:** Se aplica el mismo proceso de ordenamiento rápido a los dos subconjuntos de forma recursiva hasta que cada subconjunto tenga un solo elemento o esté vacío.

**Eficiencia:**

El ordenamiento rápido tiene una complejidad temporal promedio de O(n log n), lo que lo convierte en uno de los algoritmos de ordenamiento más eficientes.

**Ventajas:**

* Eficiencia: Es un algoritmo rápido, especialmente para conjuntos de datos grandes.
* Simplicidad: La idea central del algoritmo es simple y fácil de entender.
* Versatilidad: Se puede aplicar a una amplia variedad de estructuras de datos.

**Desventajas:**

* Peor caso: En el peor de los casos, la complejidad temporal del algoritmo puede ser O(n^2).
* Inestabilidad: No conserva el orden original de elementos iguales en el conjunto de datos.

**Estructuras de datos:**

El ordenamiento rápido se puede aplicar a diversas estructuras de datos, como:

* **Arreglos:** Es la aplicación más común del ordenamiento rápido.
* **Listas enlazadas:** Se puede aplicar, aunque la eficiencia puede verse afectada por el acceso secuencial a los elementos.
* **Árboles binarios de búsqueda:** No se suele usar con árboles binarios de búsqueda, ya que estos ya están ordenados por naturaleza.

***Ejemplo***:

public class OrdenamientoRapido {

// Ejemplo de Ordenamiento rápido en estructura de Datos

public static void main(String[] args) {

int[] numeros = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

int[] numerosOrdenados = new int[9];

int pivote = numeros[0];

int contador = 0;

// Ordenar números

for (int i = 1; i < 9; i++) {

if (numeros[i] < pivote) {

numerosOrdenados[contador] = numeros[i];

contador++;

}

}

numerosOrdenados[contador] = pivote;

contador++;

for (int i = 1; i < 9; i++) {

if (numeros[i] > pivote) {

numerosOrdenados[contador] = numeros[i];

contador++;

}

}

// Mostrar números ordenados

System.out.println("Números ordenados:");

for (int i = 0; i < 9; i++) {

System.out.print(numerosOrdenados[i] + " ");

}

System.out.println();

}

}



// Ejemplo de Ordenamiento rápido

let numeros = [9, 8, 7, 6, 5, 4, 3, 2, 1];

let numerosOrdenados = new Array(9).fill(0);

let pivote = numeros[0];

let contador = 0;

// Ordenar números

for (let i = 1; i < 9; i++) {

if (numeros[i] < pivote) {

numerosOrdenados[contador] = numeros[i];

contador++;

}

}

numerosOrdenados[contador] = pivote;

contador++;

for (let i = 1; i < 9; i++) {

if (numeros[i] > pivote) {

numerosOrdenados[contador] = numeros[i];

contador++;

}

}

// Mostrar números ordenados

console.log("Números ordenados:");

for (let i = 0; i < 9; i++) {

process.stdout.write(numerosOrdenados[i] + " ");

}

console.log();



# Ejemplo de Ordenamiento rápido

numeros = [9, 8, 7, 6, 5, 4, 3, 2, 1]

numeros\_ordenados = [0] \* 9

pivote = numeros[0]

contador = 0

# Ordenar números

for i in range(1, 9):

if numeros[i] < pivote:

numeros\_ordenados[contador] = numeros[i]

contador += 1

numeros\_ordenados[contador] = pivote

contador += 1

for i in range(1, 9):

if numeros[i] > pivote:

numeros\_ordenados[contador] = numeros[i]

contador += 1

# Mostrar números ordenados

print("Números ordenados:")

for i in range(9):

print(numeros\_ordenados[i], end=" ")

print()



### **Distribución Sort**

El ordenamiento por distribución es un algoritmo de ordenamiento eficiente para escenarios específicos. Funciona aprovechando la distribución de valores dentro del conjunto de datos.

**Funcionamiento:**

1. **Identificar el rango de valores:** Se determina el rango mínimo y máximo de valores presentes en el conjunto de datos.
2. **Crear contadores:** Se crea una lista o arreglo de contadores, con un tamaño igual al rango máximo más uno. Cada contador representa la frecuencia de un valor específico dentro del conjunto.
3. **Contar las ocurrencias:** Se recorre el conjunto de datos y se incrementa el contador correspondiente a cada valor encontrado.
4. **Calcular posiciones finales:** Se realiza un recorrido acumulativo sobre los contadores, para determinar la posición final de cada valor en el conjunto ordenado.
5. **Ordenar por distribución:** Se recorre nuevamente el conjunto de datos original, se utiliza el valor como índice en el arreglo de contadores y se coloca el elemento en la posición final calculada previamente.

**Eficiencia:**

El ordenamiento por distribución tiene una complejidad temporal promedio de O(n + k), donde n es el tamaño del conjunto de datos y k es el rango de valores. Esto lo hace muy eficiente para casos donde el rango de valores es limitado en comparación con el tamaño del conjunto.

**Ventajas:**

* Eficiente para datos con rango limitado: Es muy rápido para ordenar datos con un rango de valores acotado.
* Estable: Preserva el orden original de elementos con valores iguales.

**Desventajas:**

* Requiere espacio adicional: Necesita un arreglo adicional para almacenar los contadores, lo que puede ser un inconveniente para conjuntos de datos muy grandes.
* Ineficiente para rangos amplios: Si el rango de valores es cercano al tamaño del conjunto de datos, pierde su eficiencia.

**¿Cuándo usarlo?**

El ordenamiento por distribución es ideal para escenarios donde:

* Se tienen datos con un rango limitado de valores (por ejemplo, puntuaciones de exámenes entre 0 y 100).
* La memoria adicional no es una restricción crítica.

public class DistribucionSort {

// Ejemplo de distribución Sort

public static void main(String[] args) {

int[] numeros = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

int[] numerosOrdenados = new int[9];

int[] contador = new int[9];

int[] numerosOrdenados2 = new int[9];

int[] contador2 = new int[9];

int pivote = numeros[0];

int pivote2 = numeros[8];

int contadorPivote = 0;

int contadorPivote2 = 0;

// Ordenar números

for (int i = 0; i < 9; i++) {

if (numeros[i] < pivote) {

numerosOrdenados[contadorPivote] = numeros[i];

contadorPivote++;

}

if (numeros[i] > pivote2) {

numerosOrdenados2[contadorPivote2] = numeros[i];

contadorPivote2++;

}

}

// Mostrar números ordenados

System.out.println("Números ordenados:");

for (int i = 0; i < contadorPivote; i++) {

System.out.print(numerosOrdenados[i] + " ");

}

for (int i = 0; i < contadorPivote2; i++) {

System.out.print(numerosOrdenados2[i] + " ");

}

System.out.println();

}

}



// Ejemplo de distribución Sort

let numeros = [9, 8, 7, 6, 5, 4, 3, 2, 1];

let numerosOrdenados = new Array(9).fill(0);

let numerosOrdenados2 = new Array(9).fill(0);

let pivote = numeros[0];

let pivote2 = numeros[8];

let contadorPivote = 0;

let contadorPivote2 = 0;

// Ordenar números

for (let i = 0; i < 9; i++) {

if (numeros[i] < pivote) {

numerosOrdenados[contadorPivote] = numeros[i];

contadorPivote++;

}

if (numeros[i] > pivote2) {

numerosOrdenados2[contadorPivote2] = numeros[i];

contadorPivote2++;

}

}

// Mostrar números ordenados

console.log("Números ordenados:");

for (let i = 0; i < contadorPivote; i++) {

process.stdout.write(numerosOrdenados[i] + " ");

}

for (let i = 0; i < contadorPivote2; i++) {

process.stdout.write(numerosOrdenados2[i] + " ");

}

console.log();



# Ejemplo de distribución Sort

numeros = [9, 8, 7, 6, 5, 4, 3, 2, 1]

numeros\_ordenados = [0] \* 9

numeros\_ordenados2 = [0] \* 9

pivote = numeros[0]

pivote2 = numeros[8]

contador\_pivote = 0

contador\_pivote2 = 0

# Ordenar números

for i in range(9):

if numeros[i] < pivote:

numeros\_ordenados[contador\_pivote] = numeros[i]

contador\_pivote += 1

if numeros[i] > pivote2:

numeros\_ordenados2[contador\_pivote2] = numeros[i]

contador\_pivote2 += 1

# Mostrar números ordenados

print("Números ordenados:")

for i in range(contador\_pivote):

print(numeros\_ordenados[i], end=" ")

for i in range(contador\_pivote2):

print(numeros\_ordenados2[i], end=" ")

print()



### **Ordenamiento combsort**

Comb sort es un algoritmo de ordenamiento relativamente simple y eficiente que combina elementos del bubble sort (ordenamiento por burbuja) y del shell sort (ordenamiento por salto).

**Funcionamiento:**

1. **Gap inicial:** Se define un valor inicial para el "gap" (espacio entre elementos comparados). Este valor suele ser una fracción del tamaño del arreglo (por ejemplo, n / 1.3).
2. **Recorrido con gap:** Se recorre el arreglo comparando elementos separados por el "gap" actual. Si están en desorden, se intercambian.
3. **Reducción del gap:** Después de cada recorrido, se reduce el valor del "gap" utilizando una factor de縮小 (shusho, reducción) fijo (generalmente 1.3).
4. **Repetición:** Se repiten los pasos 2 y 3 hasta que el "gap" sea igual a 1.

**Comb Sort mejora el Bubble Sort de dos formas:**

* **Mayor salto:** En lugar de comparar elementos adyacentes, Comb Sort utiliza un "gap" más grande, permitiendo saltos mayores y un avance más rápido en el ordenamiento.
* **Eventual comparación cercana:** A medida que el "gap" se reduce, Comb Sort termina por comparar elementos cercanos, similar al Bubble Sort, para garantizar que no queden elementos desordenados.

**Eficiencia:**

Comb Sort tiene una complejidad temporal promedio de O(n log n), similar al Shell Sort. Sin embargo, en el peor de los casos, su complejidad puede degradarse a O(n^2), como el Bubble Sort.

**Ventajas:**

* Simple de implementar
* Eficiente en promedio para la mayoría de los conjuntos de datos
* Funciona bien en datos parcialmente ordenados

**Desventajas:**

* Peor caso de O(n^2)
* No es tan estable como otros algoritmos (puede alterar el orden original de elementos iguales)

**¿Cuándo usarlo?**

Comb Sort es una buena opción para situaciones donde:

* Se necesita un algoritmo de ordenamiento simple y fácil de entender.
* Se espera que los datos estén parcialmente ordenados.
* La estabilidad no es una preocupación crítica.

***Ejemplos***:

package com.example;

public class CombSort {

// Ejemplo comb sort

void sort(int arr[]) {

int n = arr.length;

// incializar el gap

int gap = n;

// inicializar el swap como true para asegurar que se ejecute al menos una vez

boolean swapped = true;

// Mantener ejecutando mientras gapes más que 1 y o la última iteración causó un

// swap

while (gap != 1 || swapped == true) {

// Encuentra el siguiente gap

gap = (gap \* 10) / 13;

if (gap < 1) {

gap = 1;

}

// verificar el swapped

swapped = false;

// Comparar todos los elementos

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

if (arr[i] > arr[i + gap]) {

// swap arr[i] y arr[i+gap]

int temp = arr[i];

arr[i] = arr[i + gap];

arr[i + gap] = temp;

// set swapped

swapped = true;

}

}

}

}

public static void main(String[] args) {

CombSort ob = new CombSort();

int arr[] = { 9, 8, 7, 6, 5, 4, 3, 2, 1 };

ob.sort(arr);

System.out.println("Números ordenados:");

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

}



// Ejemplo comb sort

function sort(arr) {

let n = arr.length;

// inicializar el gap

let gap = n;

// inicializar el swap como true para asegurar que se ejecute al menos una vez

let swapped = true;

// Mantener ejecutando mientras gap es más que 1 y la última iteración causó un swap

while (gap != 1 || swapped == true) {

// Encuentra el siguiente gap

gap = Math.floor((gap \* 10) / 13);

if (gap < 1) {

gap = 1;

}

// verificar el swapped

swapped = false;

// Comparar todos los elementos

for (let i = 0; i < n - gap; i++) {

if (arr[i] > arr[i + gap]) {

// swap arr[i] y arr[i+gap]

let temp = arr[i];

arr[i] = arr[i + gap];

arr[i + gap] = temp;

// set swapped

swapped = true;

}

}

}

}

let arr = [9, 8, 7, 6, 5, 4, 3, 2, 1];

sort(arr);

console.log("Números ordenados:");

for (let i = 0; i < arr.length; i++) {

process.stdout.write(arr[i] + " ");

}

console.log();



# Ejemplo comb sort

def sort(arr):

n = len(arr)

# inicializar el gap

gap = n

# inicializar el swap como true para asegurar que se ejecute al menos una vez

swapped = True

# Mantener ejecutando mientras gap es más que 1 y la última iteración causó un swap

while gap != 1 or swapped:

# Encuentra el siguiente gap

gap = (gap \* 10) // 13

if gap < 1:

gap = 1

# verificar el swapped

swapped = False

# Comparar todos los elementos

for i in range(n - gap):

if arr[i] > arr[i + gap]:

# swap arr[i] y arr[i+gap]

arr[i], arr[i + gap] = arr[i + gap], arr[i]

# set swapped

swapped = True

arr = [9, 8, 7, 6, 5, 4, 3, 2, 1]

sort(arr)

print("Números ordenados:")

for i in range(len(arr)):

print(arr[i], end=" ")

print()



### Fuentes:

##### Ordenamiento Burbuja

* Ordenamiento Burbuja(BubbleSort):<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoBurbuja.html>
* Wikipedia: Ordenamiento burbuja: <https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja>
* YouTube: Ordenamiento burbuja: <https://www.youtube.com/watch?v=pqZ04TT15PQ>

##### Ordenamiento aleatorio

* Wikipedia - Aleatoriedad: <https://es.wikipedia.org/wiki/Aleatoriedad>
* <https://www.geeksforgeeks.org/randomized-algorithms/>
* <https://pinetools.com/es/aleatorizar-lista>

##### Ordenamiento rápido:

* Wikipedia - Ordenamiento rápido: <https://es.wikipedia.org/wiki/Quicksort>
* GeeksforGeeks - Ordenamiento rápido: <https://www.geeksforgeeks.org/quick-sort/>

##### Distribución Sort

* Wikipedia - Ordenamiento por distribución: <https://en.wikipedia.org/wiki/Sorting_algorithm>
* GeeksforGeeks - Ordenamiento por distribución: <https://www.geeksforgeeks.org/problems/counting-sort/1>

##### CombSort

* Wikipedia - Comb Sort: <https://en.wikipedia.org/wiki/Comb_sort>
* GeeksforGeeks - Comb Sort: <https://www.geeksforgeeks.org/videos/comb-sort/>

## Link de github: <https://github.com/herculeskan/estructura_datos_act2>