**Universidad Nacional Del Altiplano**

**Facultad De Ingeniería Mecánica Eléctrica, Electrónica Y Sistemas**

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.**Escuela Profesional De Ingeniería De Sistemas**

**Practica N°5–**

**Análisis comparativo y aplicación de algoritmos de ordenamiento en estructuras lineales**

**CURSO:**

Algoritmos y Estructuras de Datos

**DOCENTE:**

Mg. Aldo Hernan Zanabria Galvez.

**ESTUDIANTE:**

Yefferson Miranda Josec

**CODIGO:** 216984

**FECHA:** 29/04/2025

**SEMESTRE:**

IV

# Introducción

El ordenamiento de datos es una operación fundamental en ciencias de la computación, con aplicaciones directas en la optimización de búsquedas, visualización de información y reducción de complejidad en otros algoritmos. Comprender cómo funcionan los algoritmos de ordenamiento y cómo se desempeñan bajo diferentes condiciones permite a los desarrolladores tomar decisiones informadas sobre su implementación en diversos contextos. Esta práctica se enfoca en el estudio comparativo de algoritmos clásicos como Bubble Sort, Selection Sort e Insertion Sort, y algoritmos más eficientes como Merge Sort y Quick Sort, utilizando estructuras de datos lineales en C++ y Python. A través de la implementación, evaluación empírica y análisis crítico, se busca formar una visión sólida y aplicada sobre los principios del ordenamiento en la programación.

# Objetivos de la práctica

El objetivo de esta práctica es implementar, analizar y comparar algoritmos de ordenamiento clásicos (Bubble Sort, Selection Sort e Insertion Sort) y eficientes (Merge Sort y Quick Sort), utilizando estructuras de datos lineales como vectores y listas en los lenguajes C++ y Python, con el fin de evaluar su rendimiento, complejidad algorítmica y adecuación según el tipo de entrada (ordenada, inversa y aleatoria). Asimismo, se busca contrastar el comportamiento de algoritmos cuadráticos frente a los eficientes mediante la medición de tiempos de ejecución y número de operaciones realizadas, y reflexionar sobre el impacto que tiene la elección de estructuras de datos en la eficiencia del proceso de ordenamiento.

# Comparación entre algoritmos

Bubble sort (c++)

#include <iostream> *// Para imprimir en consola*

#include <vector> *// Para usar std::vector*

#include <algorithm> *// Para std::swap*

#include <ctime> *// Para medir tiempo de ejecución*

#include <cstdlib> *// Para usar rand()*

*/\**

*Descripción:*

*Bubble Sort: Algoritmo de ordenamiento simple que compara e intercambia elementos adyacentes,*

*burbujeando el mayor al final en cada pasada. Es ineficiente para listas grandes.*

*\*/*

*// Función que ordena un vector usando Bubble Sort*

**void** bubbleSort(std::vector<**int**>& arr) {

**clock\_t** inicio = clock(); *// Marca el inicio del tiempo*

**int** n = arr.size(); *// Obtener el tamaño del vector*

**for** (**int** i = 0; i < n - 1; ++i) { *// Repetir n-1 veces*

**for** (std::vector<**int**>::size\_type j = 0; j < n - i - 1; ++j) { *// Usar el tipo correcto*

**if** (arr[j] > arr[j + 1]) { *// Si están fuera de orden*

std::swap(arr[j], arr[j + 1]); *// Intercambiar*

}

}

*// Mostrar el estado del vector después de cada pasada, solo los primeros 10 elementos*

**if** (i < 5) { *// Solo mostrar las primeras 5 pasadas*

std::cout << "Pasada " << i + 1 << ": ";

**for** (std::vector<**int**>::size\_type j = 0; j < 10 && j < arr.size(); ++j) {

std::cout << arr[j] << " "; *// Mostrar primeros 10 elementos*

}

std::cout << "..." << std::endl;

}

}

**clock\_t** fin = clock(); *// Marca el final del tiempo*

**double** tiempo = **double**(fin - inicio) / CLOCKS\_PER\_SEC; *// Calcular tiempo*

std::cout << "Tiempo total de ejecucion: " << tiempo << " segundos" << std::endl;

}

*// Función para generar un arreglo aleatorio de tamaño n*

std::vector<**int**> generarArregloAleatorio(**int** n) {

std::vector<**int**> arr(n);

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

arr[i] = rand() % 1000; *// Llenar con números aleatorios entre 0 y 999*

}

**return** arr;

}

*// Función principal*

**int** main() {

srand(time(0)); *// Inicializar la semilla aleatoria*

**int** n = 1000; *// Tamaño del arreglo*

*// Crear tres vectores de ejemplo*

std::vector<**int**> aleatorio = generarArregloAleatorio(n);

std::vector<**int**> ordenado(n);

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

ordenado[i] = i + 1; *// Arreglo ya ordenado*

}

std::vector<**int**> inverso(n);

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

inverso[i] = n - i; *// Arreglo en orden descendente*

}

*// Ejecutar Bubble Sort para cada tipo de vector*

std::cout << "--- ALEATORIO ---" << std::endl;

bubbleSort(aleatorio);

std::cout << "**\n**--- ORDENADO ---" << std::endl;

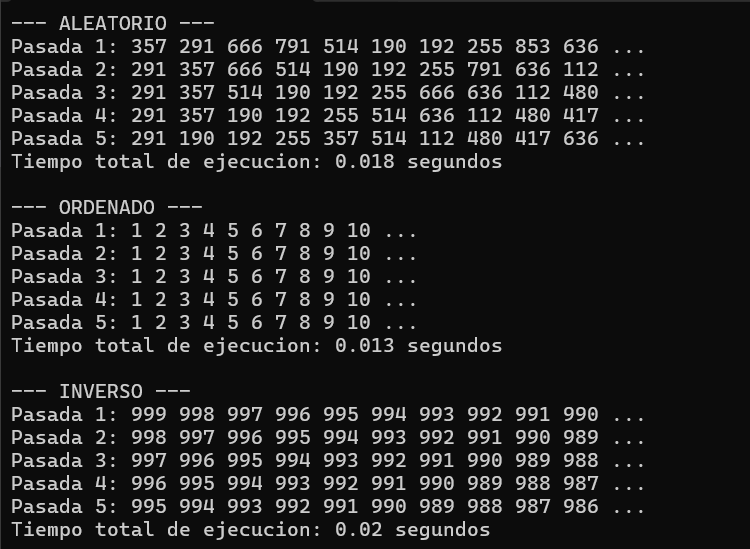
bubbleSort(ordenado);

std::cout << "**\n**--- INVERSO ---" << std::endl;

bubbleSort(inverso);

**return** 0;

}



Bubble sort (pyton)

**import** **random** *# Para generar arreglos aleatorios*

**import** **time** *# Para medir el tiempo de ejecución*

*# Función de ordenamiento Bubble Sort*

**def** bubble\_sort(arr):

n = len(arr) *# Longitud del arreglo*

**for** i **in** range(n): *# Recorrido de n veces*

**for** j **in** range(n - i - 1): *# Comparar elementos adyacentes*

**if** arr[j] > arr[j + 1]: *# Si están desordenados*

arr[j], arr[j + 1] = arr[j + 1], arr[j] *# Intercambiarlos*

**if** i < 5: *# Solo mostrar las primeras 5 iteraciones*

**print**(f"Iteración {i + 1}: {arr[:10]}...") *# Mostrar primeros 10 elementos*

*# Función para medir tiempo y mostrar resultados*

**def** probar\_bubble\_sort(nombre, arreglo):

**print**(f"**\n**--- {nombre.upper()} ---") *# Título*

copia = arreglo.copy() *# Copiar arreglo para no modificar el original*

inicio = time.time() *# Tiempo antes de ordenar*

bubble\_sort(copia) *# Ejecutar el algoritmo*

fin = time.time() *# Tiempo después de ordenar*

**print**(f"{nombre} - Tiempo: {fin - inicio:.6f} segundos") *# Mostrar tiempo*

*# Crear tres tipos de arreglos*

n = 1000

aleatorio = random.sample(range(n), n) *# Aleatorio sin repetición*

ordenado = list(range(n)) *# Arreglo ya ordenado*

inverso = list(range(n, 0, -1)) *# Arreglo en orden descendente*

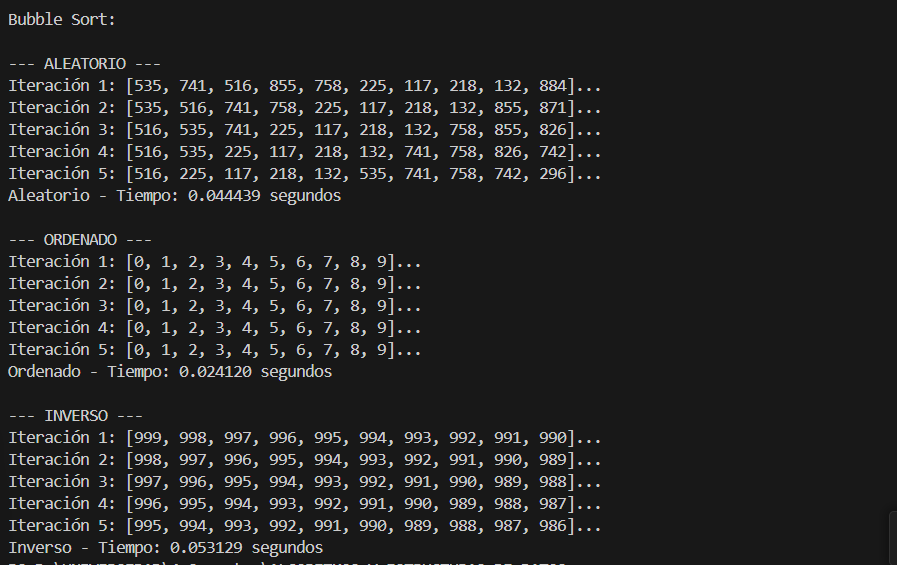
*# Ejecutar pruebas*

**print**("Bubble Sort:")

probar\_bubble\_sort("Aleatorio", aleatorio)

probar\_bubble\_sort("Ordenado", ordenado)

probar\_bubble\_sort("Inverso", inverso)



Insertion sort (c++)

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <random>

*// Algoritmo Insertion Sort*

**void** insertion\_sort(std::vector<**int**>& arr) {

**for** (**size\_t** i = 1; i < arr.size(); ++i) {

**int** key = arr[i];

**int** j = i - 1;

**while** (j >= 0 && arr[j] > key) {

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

--j;

}

arr[j + 1] = key;

**if** (i < 6) { *// Mostrar primeras 5 iteraciones*

std::cout << "Iteracion " << i << ": ";

**for** (**size\_t** k = 0; k < std::min(arr.size(), **size\_t**(10)); ++k) {

std::cout << arr[k] << " ";

}

std::cout << "...**\n**";

}

}

}

*// Función para probar el algoritmo y medir tiempo*

**void** probar\_insertion\_sort(**const** std::string& nombre, **const** std::vector<**int**>& arreglo) {

std::cout << "**\n**--- " << nombre << " ---**\n**";

std::vector<**int**> copia = arreglo;

**auto** inicio = std::chrono::high\_resolution\_clock::now();

insertion\_sort(copia);

**auto** fin = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<**double**> duracion = fin - inicio;

std::cout << nombre << " - Tiempo: " << duracion.count() << " segundos**\n**";

}

**int** main() {

**const** **int** n = 1000;

*// Crear vector aleatorio*

std::vector<**int**> aleatorio(n);

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

aleatorio[i] = i;

}

std::random\_device rd;

std::mt19937 g(rd());

std::shuffle(aleatorio.begin(), aleatorio.end(), g);

*// Crear vector ordenado*

std::vector<**int**> ordenado(n);

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

ordenado[i] = i;

}

*// Crear vector inverso*

std::vector<**int**> inverso(n);

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

inverso[i] = n - i;

}

std::cout << "Insertion Sort:**\n**";

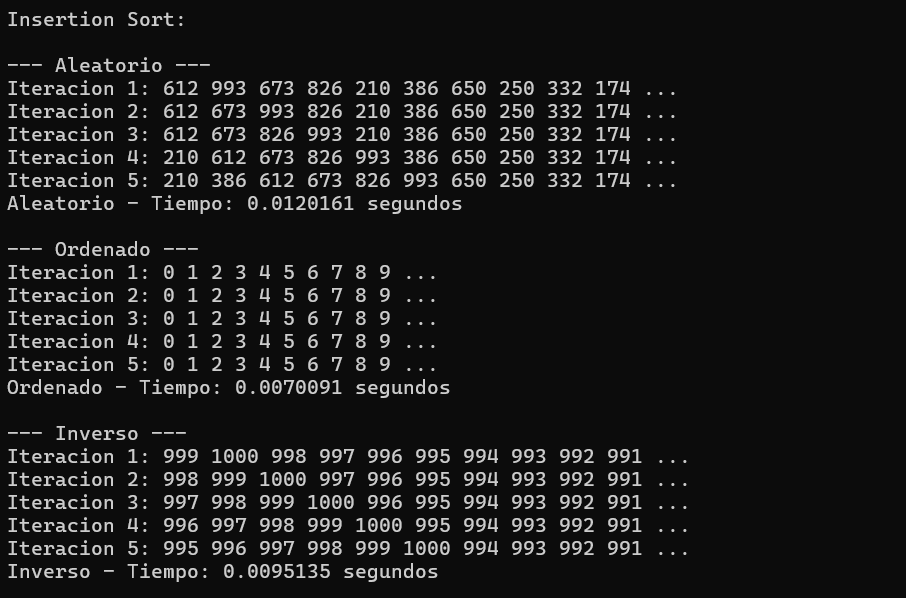
probar\_insertion\_sort("Aleatorio", aleatorio);

probar\_insertion\_sort("Ordenado", ordenado);

probar\_insertion\_sort("Inverso", inverso);

**return** 0;

}



Insertion sort (pyton)

**import** **random**

**import** **time**

*# Algoritmo Insertion Sort*

**def** insertion\_sort(arr):

**for** i **in** range(1, len(arr)): *# Comenzar desde el segundo elemento*

key = arr[i] *# Valor actual a insertar*

j = i - 1 *# Comparar con elementos anteriores*

**while** j >= 0 **and** arr[j] > key: *# Mientras haya elementos mayores*

arr[j + 1] = arr[j] *# Desplazar a la derecha*

j -= 1

arr[j + 1] = key *# Insertar en su lugar correcto*

**if** i < 6: *# Mostrar primeras 5 iteraciones*

**print**(f"Iteración {i}: {arr[:10]}...") *# Muestra parcial*

*# Función para probar el algoritmo y medir tiempo*

**def** probar\_insertion\_sort(nombre, arreglo):

**print**(f"**\n**--- {nombre.upper()} ---")

copia = arreglo.copy()

inicio = time.time()

insertion\_sort(copia)

fin = time.time()

**print**(f"{nombre} - Tiempo: {fin - inicio:.6f} segundos")

*# Generación de arreglos*

n = 1000

aleatorio = random.sample(range(n), n)

ordenado = list(range(n))

inverso = list(range(n, 0, -1))

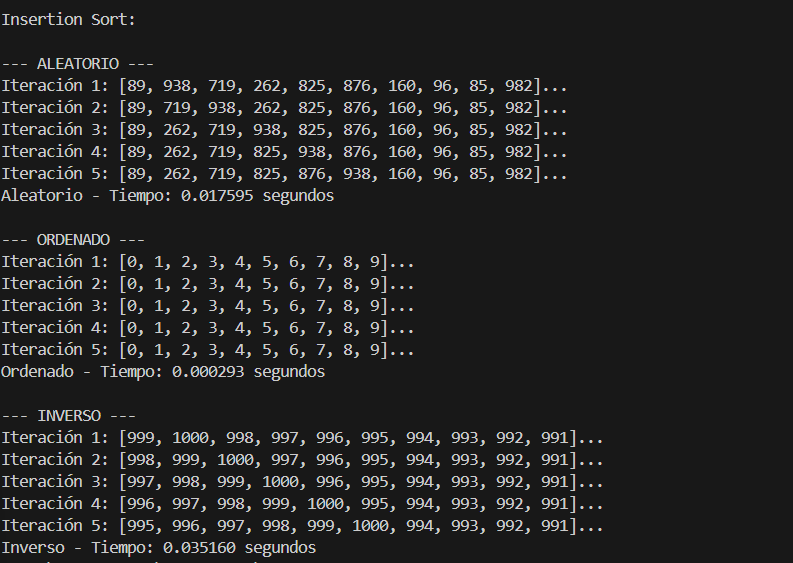
*# Ejecución*

**print**("Insertion Sort:")

probar\_insertion\_sort("Aleatorio", aleatorio)

probar\_insertion\_sort("Ordenado", ordenado)

probar\_insertion\_sort("Inverso", inverso)



Merge sort (c++)

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <random>

#include <string>

*// Algoritmo Merge Sort*

**void** merge\_sort(std::vector<**int**>& arr, **int** nivel = 0) {

**if** (arr.size() > 1) {

**int** mid = arr.size() / 2;

std::vector<**int**> L(arr.begin(), arr.begin() + mid);

std::vector<**int**> R(arr.begin() + mid, arr.end());

merge\_sort(L, nivel + 1);

merge\_sort(R, nivel + 1);

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

*// Mezcla*

**while** (i < L.size() && j < R.size()) {

**if** (L[i] < R[j]) {

arr[k++] = L[i++];

} **else** {

arr[k++] = R[j++];

}

}

*// Copiar lo que queda de L*

**while** (i < L.size()) {

arr[k++] = L[i++];

}

*// Copiar lo que queda de R*

**while** (j < R.size()) {

arr[k++] = R[j++];

}

*// Mostrar primeras fusiones*

**if** (nivel < 3) {

std::cout << "Fusion nivel " << nivel << ": ";

**for** (**size\_t** x = 0; x < std::min(**size\_t**(10), arr.size()); ++x)

std::cout << arr[x] << " ";

std::cout << "...**\n**";

}

}

}

*// Probar algoritmo y medir tiempo*

**void** probar\_merge\_sort(**const** std::string& nombre, **const** std::vector<**int**>& arreglo) {

std::cout << "**\n**--- " << nombre << " ---**\n**";

std::vector<**int**> copia = arreglo;

**auto** inicio = std::chrono::high\_resolution\_clock::now();

merge\_sort(copia);

**auto** fin = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<**double**> duracion = fin - inicio;

std::cout << nombre << " - Tiempo: " << duracion.count() << " segundos**\n**";

}

**int** main() {

**const** **int** n = 1000;

*// Aleatorio*

std::vector<**int**> aleatorio(n);

**for** (**int** i = 0; i < n; ++i) aleatorio[i] = i;

std::random\_device rd;

std::mt19937 g(rd());

std::shuffle(aleatorio.begin(), aleatorio.end(), g);

*// Ordenado*

std::vector<**int**> ordenado(n);

**for** (**int** i = 0; i < n; ++i) ordenado[i] = i;

*// Inverso*

std::vector<**int**> inverso(n);

**for** (**int** i = 0; i < n; ++i) inverso[i] = n - i;

std::cout << "Merge Sort:**\n**";

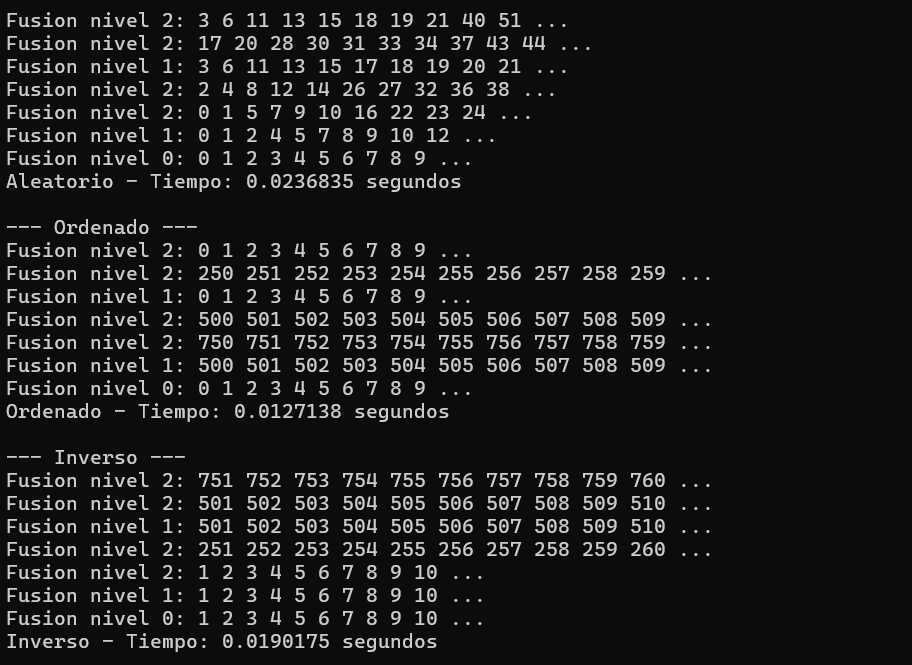
probar\_merge\_sort("Aleatorio", aleatorio);

probar\_merge\_sort("Ordenado", ordenado);

probar\_merge\_sort("Inverso", inverso);

**return** 0;

}

 Merge sort (pyton)

**import** **random**

**import** **time**

*# Algoritmo Merge Sort recursivo*

**def** merge\_sort(arr, nivel=0):

**if** len(arr) > 1:

mid = len(arr) // 2 *# Punto medio*

L = arr[:mid] *# Subarreglo izquierdo*

R = arr[mid:] *# Subarreglo derecho*

merge\_sort(L, nivel + 1) *# Ordenar mitad izquierda*

merge\_sort(R, nivel + 1) *# Ordenar mitad derecha*

i = j = k = 0 *# Índices para L, R y arr*

*# Mezclar los elementos ordenados*

**while** i < len(L) **and** j < len(R):

**if** L[i] < R[j]:

arr[k] = L[i]

i += 1

**else**:

arr[k] = R[j]

j += 1

k += 1

*# Copiar elementos restantes de L (si hay)*

**while** i < len(L):

arr[k] = L[i]

i += 1

k += 1

*# Copiar elementos restantes de R (si hay)*

**while** j < len(R):

arr[k] = R[j]

j += 1

k += 1

**if** nivel < 3: *# Mostrar solo los primeros 3 niveles*

**print**(f"Fusión nivel {nivel}: {arr[:10]}...")

*# Medir tiempo y mostrar*

**def** probar\_merge\_sort(nombre, arreglo):

**print**(f"**\n**--- {nombre.upper()} ---")

copia = arreglo.copy()

inicio = time.time()

merge\_sort(copia)

fin = time.time()

**print**(f"{nombre} - Tiempo: {fin - inicio:.6f} segundos")

*# Arreglos*

n = 1000

aleatorio = random.sample(range(n), n)

ordenado = list(range(n))

inverso = list(range(n, 0, -1))

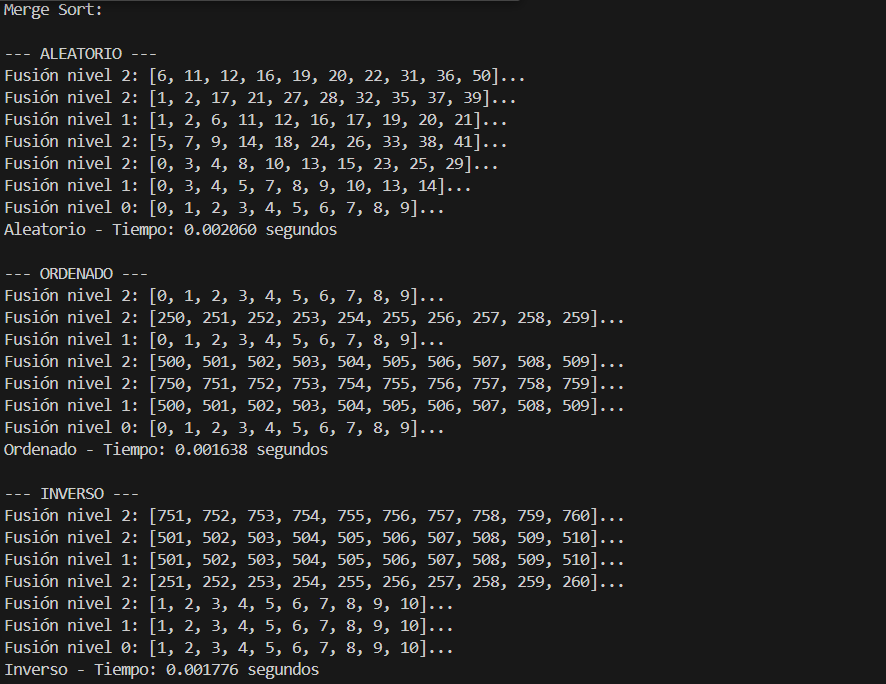
*# Ejecutar*

**print**("Merge Sort:")

probar\_merge\_sort("Aleatorio", aleatorio)

probar\_merge\_sort("Ordenado", ordenado)

probar\_merge\_sort("Inverso", inverso)



Quick sort (c ++)

#include <iostream>

#include <vector>

#include <algorithm>

#include <cstdlib>

#include <ctime>

#include <chrono>

#include <random>

**using** **namespace** std;

**using** **namespace** std::chrono;

*// Mostrar los primeros 10 elementos del segmento [low, high]*

**void** mostrarParcial(**const** vector<**int**>& arr, **int** low, **int** high) {

**int** limite = min(10, high - low + 1);

cout << "[ ";

**for** (**int** i = 0; i < limite; ++i) {

cout << arr[low + i] << " ";

}

cout << (high - low + 1 > 10 ? "... " : "") << "]";

}

*// Partición aleatoria*

**int** partition(vector<**int**>& arr, **int** low, **int** high) {

**int** pivotIndex = low + rand() % (high - low + 1);

swap(arr[pivotIndex], arr[high]);

**int** pivot = arr[high];

**int** i = low - 1;

**for** (**int** j = low; j < high; ++j) {

**if** (arr[j] <= pivot) {

++i;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

**return** i + 1;

}

*// QuickSort con control de profundidad*

**void** quickSort(vector<**int**>& arr, **int** low, **int** high, **int** nivel = 0) {

**if** (low < high) {

**int** pi = partition(arr, low, high);

**if** (nivel < 3) {

cout << "Particion nivel " << nivel << ": ";

mostrarParcial(arr, low, high);

cout << endl;

}

quickSort(arr, low, pi - 1, nivel + 1);

quickSort(arr, pi + 1, high, nivel + 1);

}

}

*// Medir y mostrar tiempo de ejecución*

**void** probarQuickSort(**const** string& nombre, **const** vector<**int**>& arreglo) {

cout << "**\n**--- " << nombre << " ---" << endl;

vector<**int**> copia = arreglo;

**auto** inicio = high\_resolution\_clock::now();

quickSort(copia, 0, copia.size() - 1);

**auto** fin = high\_resolution\_clock::now();

**auto** duracion = duration\_cast<microseconds>(fin - inicio);

cout << nombre << " - Tiempo: " << duracion.count() / 1e6 << " segundos" << endl;

}

**int** main() {

srand(time(0));

**int** n = 1000;

*// Generar vectores*

vector<**int**> aleatorio(n);

**for** (**int** i = 0; i < n; ++i) aleatorio[i] = i;

*// Barajar aleatorio*

random\_device rd;

mt19937 g(rd());

shuffle(aleatorio.begin(), aleatorio.end(), g);

vector<**int**> ordenado(n);

**for** (**int** i = 0; i < n; ++i) ordenado[i] = i;

vector<**int**> inverso(n);

**for** (**int** i = 0; i < n; ++i) inverso[i] = n - i - 1;

cout << "Quick Sort:**\n**";

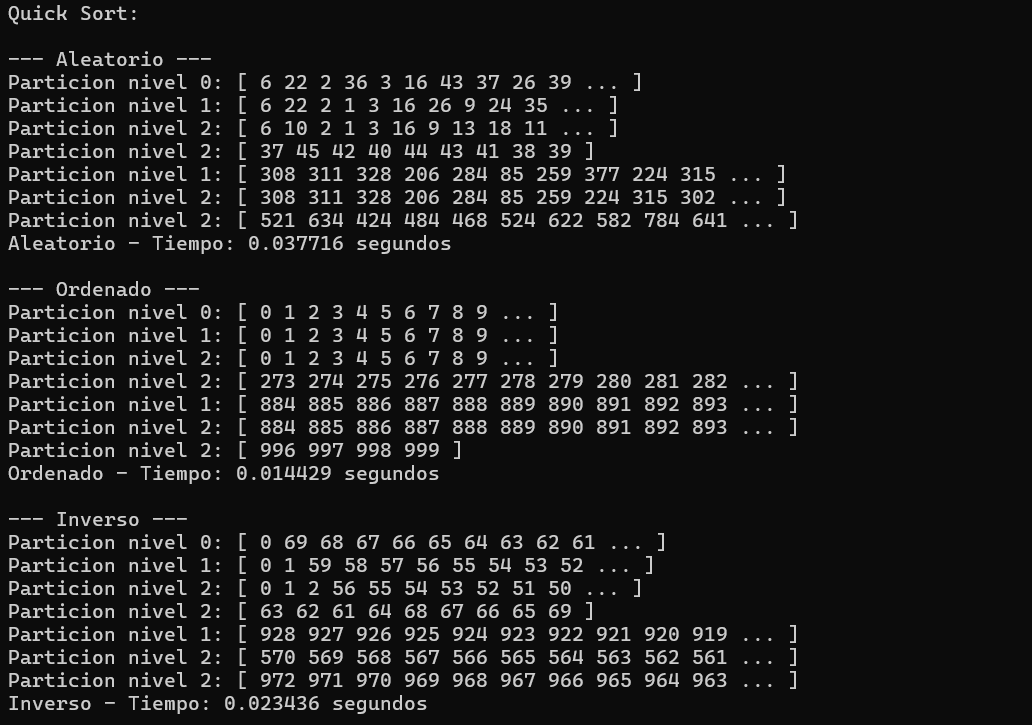
probarQuickSort("Aleatorio", aleatorio);

probarQuickSort("Ordenado", ordenado);

probarQuickSort("Inverso", inverso);

**return** 0;

}



Quick sort (pyton)

**import** **random**

**import** **time**

*# Algoritmo Quick Sort con pivote aleatorio*

**def** quick\_sort(arr, low, high, nivel=0):

**if** low < high:

pi = partition(arr, low, high)

**if** nivel < 3:

**print**(f"Partición nivel {nivel}: {arr[low:high+1][:10]}...")

quick\_sort(arr, low, pi - 1, nivel + 1)

quick\_sort(arr, pi + 1, high, nivel + 1)

*# Partición con pivote aleatorio*

**def** partition(arr, low, high):

*# Elegir índice aleatorio como pivote*

pivot\_index = random.randint(low, high)

arr[pivot\_index], arr[high] = arr[high], arr[pivot\_index] *# Mover pivote al final*

pivot = arr[high]

i = low - 1

**for** j **in** range(low, high):

**if** arr[j] <= pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

**return** i + 1

*# Medir tiempo y mostrar*

**def** probar\_quick\_sort(nombre, arreglo):

**print**(f"**\n**--- {nombre.upper()} ---")

copia = arreglo.copy()

inicio = time.time()

quick\_sort(copia, 0, len(copia) - 1)

fin = time.time()

**print**(f"{nombre} - Tiempo: {fin - inicio:.6f} segundos")

*# Arreglos*

n = 1000

aleatorio = random.sample(range(n), n)

ordenado = list(range(n))

inverso = list(range(n - 1, -1, -1))

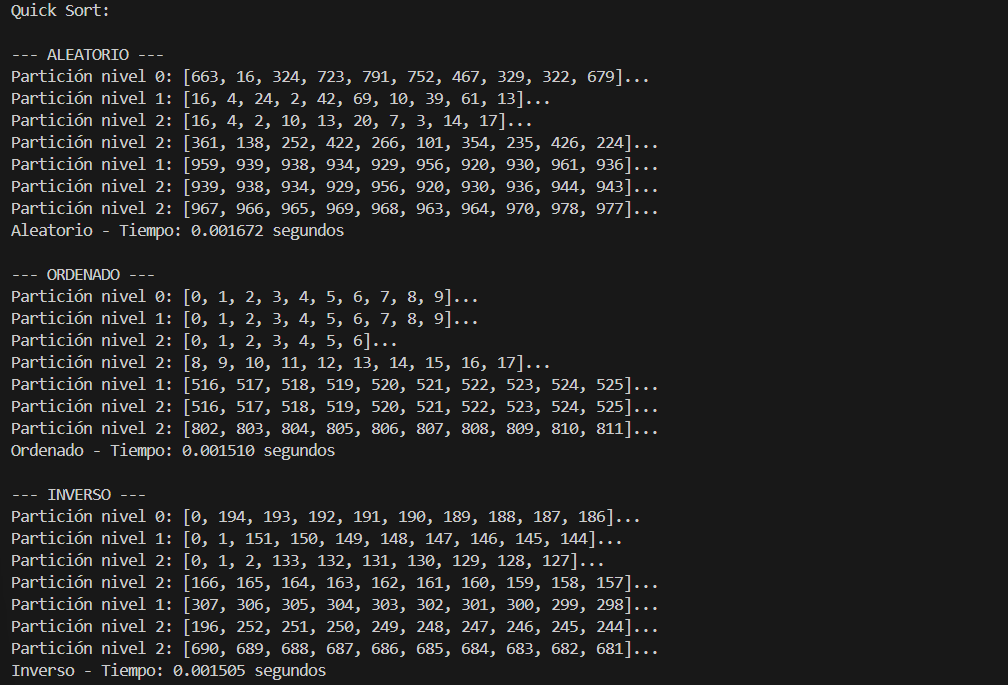
*# Ejecutar*

**print**("Quick Sort:")

probar\_quick\_sort("Aleatorio", aleatorio)

probar\_quick\_sort("Ordenado", ordenado)

probar\_quick\_sort("Inverso", inverso)



# Análisis de eficiencia y adecuación de uso

Los algoritmos de ordenamiento implementados muestran diferencias significativas en su eficiencia según el tipo de entrada y el lenguaje de programación utilizado. A continuación, se presenta un análisis detallado:

Bubble Sort:

Eficiencia: Este algoritmo tiene una complejidad computacional de O(n 2) en el peor caso y en el caso promedio, lo que lo hace ineficiente para conjuntos de datos grandes. Sin embargo, en el mejor caso (cuando el arreglo ya está ordenado), su complejidad es O(n).

Adecuación de uso: Es adecuado para conjuntos de datos pequeños o casi ordenados, donde su simplicidad puede ser una ventaja. No se recomienda para datos grandes o en aplicaciones que requieran alto rendimiento.

Insertion Sort:

Eficiencia: Al igual que Bubble Sort, su complejidad es O(n2) en el peor caso y en el caso promedio, pero O(n) en el mejor caso. Es ligeramente más eficiente que Bubble Sort en la práctica debido a que realiza menos intercambios.

Adecuación de uso: Ideal para conjuntos de datos pequeños o parcialmente ordenados. Es útil en situaciones donde los datos se reciben de forma secuencial y necesitan ser insertados en su posición correcta.

Merge Sort:

Eficiencia: Tiene una complejidad de O(nlogn) en todos los casos (peor, promedio y mejor), lo que lo hace muy eficiente para conjuntos de datos grandes.

Adecuación de uso: Es adecuado para aplicaciones que requieren estabilidad (mantener el orden relativo de elementos iguales) y un rendimiento consistente, como en bases de datos o sistemas de archivos.

Quick Sort:

Eficiencia: En el caso promedio, su complejidad es O(n log n), pero en el peor caso (por ejemplo, cuando el arreglo ya está ordenado o inversamente ordenado), puede degradarse a O(n2)O(n2). Sin embargo, con la elección de un pivote aleatorio, se reduce la probabilidad del peor caso.

Adecuación de uso: Es ideal para conjuntos de datos grandes y aleatorios, donde su velocidad promedio lo hace superior a otros algoritmos. No es estable, por lo que no es recomendable cuando el orden relativo es importante.

# Conclusiones personales

1. Elección del algoritmo: La elección del algoritmo de ordenamiento debe basarse en el tamaño y la naturaleza de los datos. Para conjuntos pequeños, algoritmos simples como Bubble Sort o Insertion Sort pueden ser suficientes, mientras que para conjuntos grandes, Merge Sort o Quick Sort son más adecuados.
2. Impacto del lenguaje: Se observó que las implementaciones en C++ tienden a ser más rápidas que en Python debido a la naturaleza compilada de C++ frente a la interpretada de Python. Esto es especialmente notable en algoritmos con alta complejidad computacional.
3. Optimización: La optimización del código, como la elección de estructuras de datos eficientes y la reducción de operaciones redundantes, puede mejorar significativamente el rendimiento, incluso en algoritmos con alta complejidad.

# Complejidad Computacional

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmo | Mejor caso | Caso promedio | Peor caso |
| Bubble Sort | O(n) | O(n 2) | O(n 2) |
| Insertion Sort | O(n) | O(n 2) | O(n 2) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| Quick Sort | O(n log n) | O(n log n) | O(n 2) |

# Comparación técnica esperada

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algoritmo** | **Tipo de entrada** | **Tiempo c++** | **Tiempo Python** | **Comparaciones** | **Intercambios** |
| Bubble sort | Aleatoria | 0.018 segundos | 0.0127138 segundos | Multiples | Multiples |
| Merge Sort | Ordenada | 0.024120 segundos | 0.0016 38 segundos | Menos que buble sort | No aplica (Estable) |
| Quick Sort | Inversa | 0.023436 segundos | 0.001505 segundos | Depende del pivote | Variable |

# Enlace a OnlineGDB / Google Colab o GitHub (si aplica)

<https://github.com/yefferson12355/Algoritmos-y-Estructuras-de-Datos/tree/master/ACTIVIDADES/Actividad%20N%C2%B05>

# Ensayo de Metodos de Ordenamiento en Programacion

Ensayo sobre Métodos de Ordenamiento en Programación

# Introducción

En el ámbito de la programación y la informática, el ordenamiento de datos es una operación fundamental. Ordenar listas o arreglos permite realizar búsquedas más eficientes, optimizar algoritmos, y facilita la visualización y el análisis de información. Existen numerosos algoritmos de ordenamiento, y se pueden clasificar principalmente en dos grandes grupos: algoritmos basados en comparación y algoritmos no basados en comparación. Cada uno de estos algoritmos tiene características propias, ventajas y desventajas, y su uso depende del tipo y volumen de datos, así como de los requisitos de eficiencia y recursos computacionales.

# Algoritmos de Ordenamiento Basados en Comparación

Los algoritmos de ordenamiento basados en comparación funcionan comparando pares de elementos dentro de la lista. La base de estos métodos es que se evalúan dos elementos para determinar cuál debe ir antes en el orden.

# 1. Bubble Sort

Bubble Sort es uno de los algoritmos más sencillos. Funciona iterando repetidamente por la lista, comparando elementos adyacentes y cambiándolos si están en el orden incorrecto.

- Ventajas: Fácil de implementar.

- Desventajas: Muy ineficiente para listas grandes. Tiene una complejidad de tiempo de O(n²).

- Uso típico: Casos educativos o listas muy pequeñas.

# 2. Selection Sort

Este algoritmo selecciona el elemento más pequeño (o más grande, según el orden) de la lista y lo coloca en su posición final correcta, repitiendo el proceso con los elementos restantes.

- Ventajas: Implementación sencilla, realiza el mínimo número posible de intercambios.

- Desventajas: Igual que Bubble Sort, tiene una complejidad de O(n²).

- Uso típico: Situaciones donde el costo de escribir en memoria es elevado, ya que realiza menos intercambios.

# 3. Insertion Sort

Funciona construyendo la lista ordenada de izquierda a derecha, tomando cada nuevo elemento y colocándolo en su posición correcta dentro de los elementos ya ordenados.

- Ventajas: Eficiente para listas pequeñas o casi ordenadas.

- Desventajas: Ineficiente para listas grandes no ordenadas (O(n²)).

- Uso típico: Pequeños subconjuntos de datos, como en algoritmos híbridos.

# 4. Merge Sort

Es un algoritmo basado en la técnica de divide y vencerás. Divide el arreglo en mitades, ordena recursivamente cada mitad y luego las combina.

- Ventajas: Complejidad de tiempo O(n log n) en todos los casos.

- Desventajas: Requiere memoria adicional proporcional al tamaño de la lista.

- Uso típico: Ordenamiento de listas grandes donde la eficiencia es importante.

# 5. Quick Sort

También usa la técnica de divide y vencerás, eligiendo un pivote, particionando el arreglo en torno a él y ordenando recursivamente las particiones.

- Ventajas: Muy rápido en la práctica, con complejidad promedio de O(n log n).

- Desventajas: Su peor caso es O(n²), aunque puede mitigarse con pivotes aleatorios.

- Uso típico: Generalmente utilizado en librerías estándar por su velocidad.

# 6. Heap Sort

Convierte el arreglo en una estructura de datos llamada heap, y luego extrae el máximo (o mínimo) sucesivamente para construir la lista ordenada.

- Ventajas: Complejidad de O(n log n), no requiere espacio adicional significativo.

- Desventajas: Más lento en la práctica que Quick Sort.

- Uso típico: Donde se requiere ordenamiento en el lugar y buena eficiencia en el peor caso.

# Algoritmos de Ordenamiento No Basados en Comparación

Los algoritmos no basados en comparación ordenan los datos sin utilizar comparaciones entre elementos. En su lugar, se basan en propiedades matemáticas o características internas de los datos. Son muy eficientes cuando se cumplen ciertas condiciones.

# 1. Counting Sort

Counting Sort es un algoritmo que cuenta la cantidad de veces que aparece cada valor en el conjunto de datos y utiliza esta información para colocar los elementos en orden.

- Requisitos: Solo funciona con datos enteros no negativos dentro de un rango conocido.

- Complejidad: O(n + k), donde n es el número de elementos y k es el rango de valores.

- Ventajas: Extremadamente rápido para datos con un rango limitado.

- Desventajas: No es eficiente si el rango de los datos (k) es muy grande. No es un algoritmo en el lugar (requiere memoria adicional).

- Uso típico: Ordenar edades, puntuaciones o valores discretos pequeños.

# 2. Radix Sort

Radix Sort ordena los elementos procesando cada dígito individual, desde el menos significativo hasta el más significativo. Utiliza una técnica como Counting Sort como subrutina.

- Requisitos: Funciona bien con enteros o cadenas que pueden descomponerse en componentes ordenables (como dígitos o caracteres).

- Complejidad: O(nk), donde k es el número de dígitos.

- Ventajas: Puede superar el límite teórico de O(n log n) de los algoritmos de comparación cuando los datos cumplen las condiciones necesarias.

- Desventajas: Requiere memoria adicional, y puede no ser práctico para valores con muchos dígitos o gran variabilidad.

- Uso típico: Ordenar códigos postales, números telefónicos, claves numéricas largas.

# Comparación entre Algoritmos Basados y No Basados en Comparación

|  |  |  |
| --- | --- | --- |
| **Característica** | **Basados en Comparación** | **No Basados en Comparación** |
| Complejidad Mínima Teórica | | O(n log n) | O(n) en ciertos casos |
| Comparación de Elementos | | Sí | | No |
| Aplicabilidad General | | Amplia (cualquier tipo de datos) | Limitada (enteros, cadenas |
| Uso de Memoria | | Bajo o moderado | Puede requerir memoria adicional |
| Ejemplos | QuickSort, MergeSort | Counting Sort, Radix Sort |

# Consideraciones Prácticas

- Tamaño del dataset: Para conjuntos pequeños, algoritmos como Insertion Sort pueden ser más rápidos.

- Memoria disponible: Algoritmos como Merge Sort y Counting Sort requieren memoria extra.

- Distribución de los datos: Algunos algoritmos se benefician de listas ya parcialmente ordenadas (como Insertion Sort).

- Estabilidad: Algoritmos estables preservan el orden relativo de elementos iguales. Esto es útil en ordenamientos múltiples (por ejemplo, por apellido y luego por nombre).

# Conclusión

El ordenamiento de datos es una de las tareas más fundamentales en la programación. Existen múltiples algoritmos con diferentes propiedades y niveles de eficiencia. Los algoritmos basados en comparación, como Quick Sort y Merge Sort, son versátiles y aplicables a una gran variedad de casos. En cambio, los algoritmos no basados en comparación, como Counting Sort y Radix Sort, ofrecen una eficiencia superior en contextos específicos, como cuando se trabaja con enteros en un rango conocido. Elegir el algoritmo adecuado depende del contexto del problema, el tipo de datos, los recursos disponibles y los requisitos del sistema. Conocer a fondo estas herramientas es esencial para diseñar software eficiente y robusto