**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

/\*

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** (**int** j = **0**; j < n - i - **1**; ++j) { // Recorrer hasta el último no ordenado

**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

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

**for** (**int** val : arr)

std::cout << val << " ";

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 de ejecucion: " << tiempo << " segundos" << std::endl;

}

}

// Función principal

**int** **main**() {

// Crear un vector con elementos desordenados

std::vector<**int**> numeros = {**64**, **34**, **25**, **12**, **22**, **11**, **90**};

std::cout << "Vector original: ";

**for** (**int** n : numeros)

std::cout << n << " ";

std::cout << std::endl;

// Aplicar Bubble Sort

bubbleSort(numeros);

// Mostrar el vector ordenado

std::cout << "Vector ordenado: ";

**for** (**int** n : numeros)

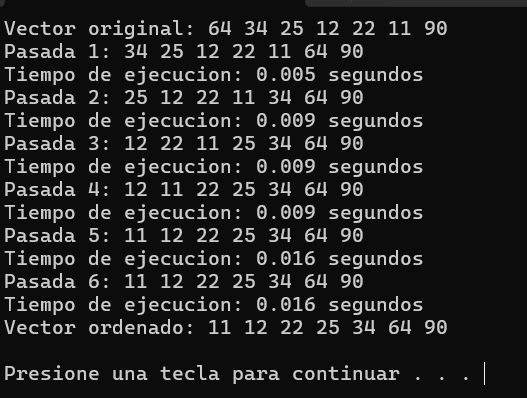
std::cout << n << " ";

std::cout << std::endl;

**return** **0**;

}

// Fin del código



Bubble sort (pyton)

**import** **time** # Importa el módulo time para medir el tiempo de ejecución

**import** **random** # Importa random para generar números aleatorios

# Algoritmo Bubble Sort

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

n = len(arr) # Obtiene el tamaño del arreglo (lista)

**for** i **in** range(n - **1**): # Bucle externo: hace n-1 pasadas

# Medir tiempo al inicio de la pasada

pasada\_inicio = time.time()

**for** j **in** range(n - i - **1**): # Bucle interno: compara elementos no ordenados

**if** arr[j] > arr[j + **1**]: # Si el actual es mayor que el siguiente

arr[j], arr[j + **1**] = arr[j + **1**], arr[j] # Intercambia los elementos

# Medir tiempo al final de la pasada

pasada\_fin = time.time()

# Mostrar estado y duración de la pasada

**print**(f"Pasada {i + 1}: {arr} - Duración: {pasada\_fin - pasada\_inicio:.6f} segundos")

# Función para probar Bubble Sort con un arreglo y medir su tiempo

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

copia = arreglo.copy() # Crea una copia para no modificar el arreglo original

inicio = time.time() # Marca el inicio de la ejecución

bubble\_sort(copia) # Aplica el algoritmo de ordenamiento

fin = time.time() # Marca el final

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

# 1. Lista de 1000 números aleatorios entre 1 y 10000

aleatorio = [random.randint(**1**, **10000**) **for** \_ **in** range(**1000**)]

# 2. Lista ordenada de menor a mayor (mejor caso para algunos algoritmos)

ordenado = sorted(aleatorio) # 'sorted()' es una función de Python que devuelve una nueva lista ordenada

# 3. Lista ordenada de mayor a menor (peor caso para Bubble Sort, por ejemplo)

inverso = sorted(aleatorio, reverse=True) # Ordena al revés con reverse=True

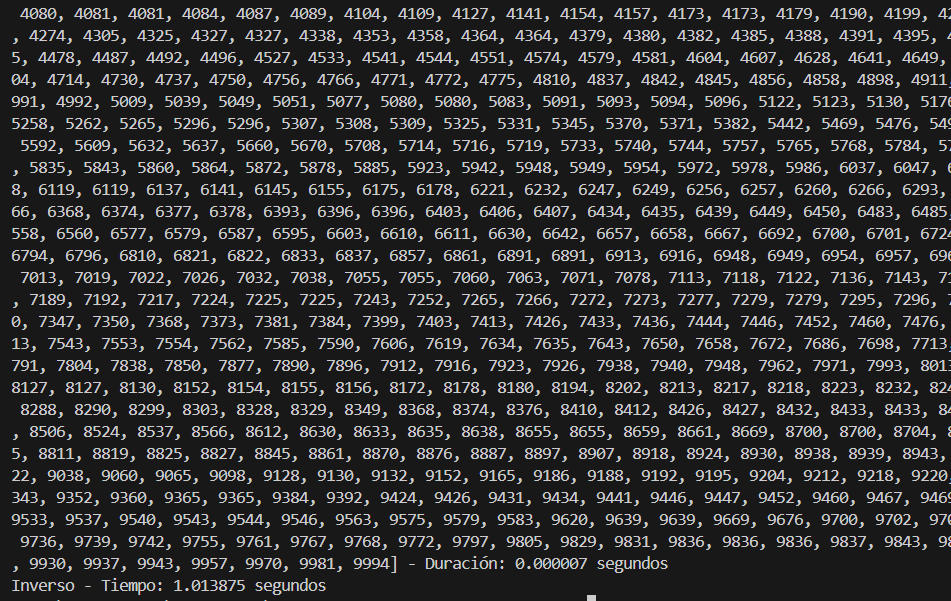
# Ejecuta la prueba con los tres tipos de lista

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

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

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

...



…El Resto de las implementaciones en el enlace de Git Hub

# Análisis de eficiencia y adecuación de uso

# Conclusiones personales (mínimo tres)

# Complejidad Computacional

# V. Comparación técnica esperada

Algoritmo Tipo de

entrada

Tiempo C++

(ms)

Tiempo

Python (ms)

Comparaciones Intercambios

Bubble

Sort

Aleatoria

Merge

Sort

Ordenada

Quick Sort Inversa

# 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>

Algoritmos No Basados en Comparación:

Algoritmos No Basados en Comparación: Utilizan propiedades específicas de los datos

para ordenar sin comparaciones directas, como Counting Sort y Radix Sort.