EJERCICIO 1

Curso.h

```
#pragma once

#include <iostream>
#include <string>

struct Curso {
    int id;
    std::string nombre;

    bool operator<(const Curso& other) const {
        return id < other.id;
    }

    bool operator>(const Curso& other) const {
        return id > other.id;
    }

    void print() const {
        std::cout << "[ID: " << id << ", Nombre: " << nombre << "]";
    }
};
```

ListaSimple.h

```
#pragma once

#include "Curso.h"
#include <iomanip> // Para formato de tabla

// Definición del nodo de la lista
struct Nodo
{
    Curso data;
    Nodo *next;

    // Constructor
    Nodo(const Curso &c) : data(c), next(nullptr) {}
};

class ListaSimple
{
private:
    Nodo *head;

    // Variables de análisis
    int totalComparaciones = 0;
    int totalPunterosModificados = 0;

    bool insertarOrdenado(Nodo *&sortedHead, Nodo *key)
```

```
    {
        if (!sortedHead || key->data < sortedHead->data)
        {
            key->next = sortedHead;
            sortedHead = key;
            totalPunterosModificados += 2;
            return true;
        }

        Nodo *current = sortedHead;
        Nodo *prev = nullptr;
        int compCount = 0;
        int dispCount = 0;

        while (current != nullptr && current->data < key->data)
        {
            prev = current;
            current = current->next;
            compCount++;
        }
        totalComparaciones += compCount;

        if (prev == nullptr)
        {
            return false;
        }

        key->next = current;
        prev->next = key;
        totalPunterosModificados += 2;

        std::cout << "  -> Inserción en posición: ";
        if (current)
            current->data.print();
        else
            std::cout << "FINAL";
        std::cout << "\n";

        return true;
    }

public:
    ListaSimple() : head(nullptr) {}
    ~ListaSimple()
    {
        // Liberación de memoria (Destructor)
        Nodo *current = head;
        while (current != nullptr)
        {
            Nodo *next = current->next;
            delete current;
```

```
            current = next;
        }
        head = nullptr;
    }

    void add(int id, const std::string &nombre)
    {
        Curso c = {id, nombre};
        Nodo *newNode = new Nodo(c);
        newNode->next = head;
        head = newNode;
    }

    void print() const
    {
        Nodo *current = head;
        if (!current)
        {
            std::cout << "Lista vacía.\n";
            return;
        }
        while (current != nullptr)
        {
            current->data.print();
            if (current->next)
            {
                std::cout << " -> ";
            }
            current = current->next;
        }
        std::cout << " -> NULL\n";
    }

    void insertionSort()
    {
        if (!head || !head->next)
        {
            return;
        }

        Nodo *sorted = nullptr;
        Nodo *current = head;

        std::cout <<
"\n=======================================================\n";
        std::cout << "       INICIO DE ORDENAMIENTO (INSERTION SORT)       \n";
        std::cout <<
"=======================================================\n";

        while (current != nullptr)
        {
```

```
            Nodo *key = current;
            current = current->next;
            key->next = nullptr;

            std::cout << "\n--- CLAVE: ";
            key->data.print();
            std::cout << " ---\n";

            int initialPointers = totalPunterosModificados;

            if (insertarOrdenado(sorted, key))
            {
                std::cout << "  -> Nodos desplazados: 1 (el nodo clave)\n";
            }
            else
            {
                std::cout << "  -> Nodos desplazados: 0 (el nodo clave ya
estaba en orden)\n";
            }

            head = sorted;

            std::cout << "  -> Comparaciones realizadas en esta pasada: " <<
(totalComparaciones - initialPointers / 2) << "\n";
            std::cout << "  -> Punteros modificados en esta pasada: " <<
(totalPunterosModificados - initialPointers) << "\n";

            std::cout << "\nEstado de la Lista: \n";
            print();
            std::cout <<
"-----------------------------------------------------\n";
        }
    }

    void showFinalAnalysis() const
    {
        std::cout <<
"\n=====================================================\n";
        std::cout << "             ANÁLISIS FINAL DEL ALGORITMO
\n";
        std::cout <<
"=====================================================\n";

        std::cout << std::setw(30) << std::left << "Total de Comparaciones:" <<
totalComparaciones << "\n";
        std::cout << std::setw(30) << std::left << "Total de Punteros
Modificados:" << totalPunterosModificados << "\n";
    }
};
```

Main.cpp

```
#include <iostream>
#include <locale.h>
#include "ListaSimple.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    ListaSimple cursos;

    cout << "--- Construcción de la Lista (Ingreso de Cursos) ---\n";

    cursos.add(402, "Física 2");
    cursos.add(203, "EDO");
    cursos.add(501, "Metodología Investigativa");
    cursos.add(101, "Matemática");
    cursos.add(315, "Cultura Ambiental");

    cout << "Lista Original (Desordenada):\n";
    cursos.print();
    cout << "-----------------------------------------------------\n";

    cursos.insertionSort();

    cout << "\nLista Final Ordenada por ID:\n";
    cursos.print();

    // 3. Presentar el análisis final
    cursos.showFinalAnalysis();

    return 0;
}

 /*Razones por las que Insertion Sort es adecuado para Listas Enlazadas:
    Bajo Costo de Desplazamiento: Insertion Sort solo requiere cambiar
punteros, lo que es una operación O(1)[cite: 3, 11].
    No se necesita mover la estructura de datos completa (como ocurre en
arreglos, donde un desplazamiento es O(N)).\n";

    No requiere acceso aleatorio (índice):** Los algoritmos que dependen de
índices (como Quicksort o Heapsort en arreglos)
    son lentos en listas enlazadas, ya que acceder a un índice 'i' requiere
recorrer la lista O(i). Insertion Sort opera
    secuencialmente, lo cual es natural para esta estructura.\n";

    Eficiente para listas casi ordenadas: Si la lista ya está casi ordenada,
Insertion Sort realiza pocos desplazamientos
    y comparaciones, lo que lo hace muy rápido en el mejor caso (O(N));*/
```

EJERCICIO 2

Transaccion.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>

struct Transaccion {
    std::string id_transaccion;
    double monto;
    std::string tipo;
    std::string fecha;

    int order;

    void imprimir() const {
        std::cout << "|" << std::setw(5) << std::left << order
                  << "|" << std::setw(15) << std::left << id_transaccion
                  << "|" << std::setw(8) << std::right << std::fixed <<
std::setprecision(2) << monto
                  << "|" << std::setw(10) << std::left << tipo
                  << "|" << std::setw(10) << std::left << fecha << "|";
    }
};

template <typename T>
void imprimirArray(T arr[], int n) {
    std::cout << "+-----+---------------+--------+----------+----------+\n";
    std::cout << "|Ord. | ID Transacción| Monto  | Tipo     | Fecha    |\n";
    std::cout << "+-----+---------------+--------+----------+----------+\n";
    for (int i = 0; i < n; ++i) {
        arr[i].imprimir();
        std::cout << "\n";
    }
    std::cout << "+-----+---------------+--------+----------+----------+\n";
}

template <typename T>
void insertionSort(T arr[], int n) {
    int totalComparaciones = 0;
    int totalDesplazamientos = 0;

    std::cout <<
"\n=================================================================\n";
    std::cout << "      INICIO DE ORDENAMIENTO POR MONTO (INSERTION SORT
ESTABLE)    \n";
    std::cout <<
"=================================================================\n";

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

```
        T key = arr[i];
        int j = i - 1;
        int compCount = 0;
        int dispCount = 0;

        std::cout << "\n--- Iteración " << i << " (Clave: ";
        key.imprimir();
        std::cout << ") ---\n";

        while (j >= 0 && arr[j].monto > key.monto) {
            arr[j + 1] = arr[j];
            j = j - 1;
            compCount++;
            dispCount++;
        }
        totalComparaciones += compCount;
        totalDesplazamientos += dispCount;

        if (j + 1 != i) {
            arr[j + 1] = key;
        }

        std::cout << "  -> Comparaciones efectuadas: " << compCount << "\n"; //
        std::cout << "  -> Desplazamientos realizados: " << dispCount <<
"\n"; //
        std::cout << "  -> Estado del arreglo tras esta iteración: \n"; //
        imprimirArray(arr, n);
    }

    std::cout <<
"\n================================================================\n";
    std::cout << "               ANÁLISIS FINAL Y ESTABILIDAD\n";
    std::cout <<
"================================================================\n";
    std::cout << "Total de Comparaciones: " << totalComparaciones << "\n";
    std::cout << "Total de Desplazamientos (movimiento de registros): " <<
totalDesplazamientos << "\n";


}
```

main.cpp

```
#include <iostream>
#include <locale.h>
#include "Transaccion.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");
```

```
    // Arreglo de transacciones bancarias. Se incluyen montos iguales ($100.00)
    // para demostrar la estabilidad del algoritmo.
    Transaccion transactions[] = {
        // ID     Monto   Tipo    Fecha       Order (Original)
        {"T105", 250.00, "Dep.", "2025-11-24", 1},
        {"T201", 100.00, "Ret.", "2025-11-25", 2}, // Monto igual: 100.00
(Order 2)
        {"T100", 750.50, "Dep.", "2025-11-25", 3},
        {"T310", 100.00, "Com.", "2025-11-24", 4}, // Monto igual: 100.00
(Order 4)
        {"T050", 400.00, "Ret.", "2025-11-23", 5}
    };

    int n = sizeof(transactions) / sizeof(transactions[0]);

    cout << "--- Estado Inicial del Arreglo de Transacciones ---\n";
    imprimirArray(transactions, n);

    // Aplicar Insertion Sort (ordenando por monto)
    insertionSort(transactions, n);

    cout << "\n--- Estado Final del Arreglo (Ordenado por Monto) ---\n";
    imprimirArray(transactions, n);

    cout << "\nVerificación de Estabilidad:\n";
    cout << "Las transacciones con Monto = $100.00 (T201 y T310) mantuvieron su
orden original:\n";
    cout << "  - T201 (Order 2) aparece antes que T310 (Order 4).\n";
    cout << "Esto confirma la estabilidad del Insertion Sort.\n";


    return 0;
}

/*Preservación de la Estabilidad (monto igual):
 El algoritmo Insertion Sort es naturalmente estable porque:
 La condición de comparación es estricta: `arr[j].monto > key.monto`.

 Si dos elementos tienen el mismo monto (`arr[j].monto == key.monto`), la
condición es falsa,
 y la clave (`key`) se inserta después del elemento igual (`arr[j]`).

 Esto garantiza que si el elemento A estaba antes que el elemento B en el
arreglo original y
 `monto(A) == monto(B)`, A seguirá estando antes que B en el arreglo ordenado,
cumpliendo con la definición de estabilidad.

Implicaciones en aplicaciones reales:
Clasificación Secundaria: Si ordenas por 'monto' y luego por 'fecha' (en el
orden original),
```

un algoritmo estable asegura que las transacciones con el mismo monto mantengan el orden de 'fecha' que tenían antes.
Auditoría: Permite al sistema mantener el orden cronológico o de ingreso (nuestro campo `order`)
dentro de grupos de datos idénticos.*/

EJERCICIO 3

Producto.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>
#include <vector>

struct Producto {
    int codigo;
    std::string descripcion;
    std::string categoria;
    double peso;
    double precio;

    void print() const {
        std::cout << "|" << std::setw(8) << std::left << codigo
                  << "|" << std::setw(15) << std::left << descripcion
                  << "|" << std::setw(12) << std::left << categoria
                  << "|" << std::setw(7) << std::right << std::fixed <<
std::setprecision(2) << peso
                  << "|" << std::setw(7) << std::right << std::fixed <<
std::setprecision(2) << precio << "|";
    }
};

template <typename T>
void printArray(const std::vector<T>& arr) {
    std::cout << "+--------+---------------+------------+-------+-------+\n";
    std::cout << "| Código | Descripción   | Categoría  | Peso  | Precio|\n";
    std::cout << "+--------+---------------+------------+-------+-------+\n";
    for (const auto& item : arr) {
        item.print();
        std::cout << "\n";
    }
    std::cout << "+--------+---------------+------------+-------+-------+\n";
}


int totalComparaciones = 0;

bool esMenor(const Producto& prod1, const Producto& prod2, int& compCount) {
```

```
    compCount++;
    if (prod1.categoria != prod2.categoria) {
        return prod1.categoria < prod2.categoria;
    }

    compCount++;
    if (prod1.precio != prod2.precio) {

        return prod1.precio < prod2.precio;
    }

    compCount++;
    return prod1.peso < prod2.peso;
}

template <typename T>
void selectionSort(std::vector<T>& arr) {
    int n = arr.size();

    std::cout <<
"\n========================================================================\n";
    std::cout << "      INICIO DE ORDENAMIENTO (SELECTION SORT CON CRITERIO
COMPUESTO)    \n";
    std::cout <<
"========================================================================\n";

    for (int i = 0; i < n - 1; ++i) {
        int min_idx = i;
        int compCount = 0;
        bool intercambioOcurrido = false;

        std::cout << "\n--- Pasada " << (i + 1) << " (Buscando el mínimo desde
el índice " << i << ") ---\n";

        for (int j = i + 1; j < n; ++j) {
            // Implementar la comparación compuesta
            if (esMenor(arr[j], arr[min_idx], compCount)) {
                min_idx = j; // Nueva posición del mínimo
            }
        }
        totalComparaciones += compCount;

        std::cout << "  -> Búsqueda del Mínimo: " << compCount << "
comparaciones necesarias.\n";
        std::cout << "  -> La posición del Mínimo encontrado es: " << min_idx
<< ".\n";

        if (min_idx != i) {
            std::swap(arr[i], arr[min_idx]);
            intercambioOcurrido = true;
        }
```

```
        std::cout << "  -> ¿Ocurrió un intercambio? " << (intercambioOcurrido ?
"SÍ" : "NO") << ".\n";

        std::cout << "\nEstado del Vector tras la Pasada " << (i + 1) << ":\n";
        printArray(arr);
    }

    // Análisis final
    std::cout <<
"\n========================================================================\n";
    std::cout << "                           ANÁLISIS FINAL\n";
    std::cout <<
"========================================================================\n";
    std::cout << "Total de Comparaciones (Contando los 3 niveles): " <<
totalComparaciones << "\n";
    std::cout << "Total de Intercambios realizados: " << n - 1 << " (Siempre
ocurre uno por pasada, a menos que el mínimo ya esté en su lugar, pero la
lógica de Selection Sort es hacer a lo sumo N-1 intercambios).\n";

}
```

main.cpp

```
#include <iostream>
#include <locale.h>
#include <vector>
#include "Producto.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    std::vector<Producto> productos = {
        {101, "Leche", "Lacteo", 1.05, 3.50},
        {204, "Jugo Naranja", "Bebida", 0.50, 2.00},
        {310, "Yogurt", "Lacteo", 0.30, 1.20},
        {402, "Queso", "Lacteo", 0.40, 1.20},
        {507, "Agua 1L", "Bebida", 1.10, 2.00}
    };

    cout << "--- Estado Inicial del Vector de Productos ---\n";
    printArray(productos);

    selectionSort(productos);

    cout << "\n--- Estado Final del Vector (Ordenado: Categoría -> Precio ->
Peso) ---\n";
    printArray(productos);
```

```
    return 0;
}

/*Selection Sort es ideal para registros 'pesados' (grandes estructuras) porque
minimiza el costo de movimiento.

Mínimo Movimiento: Realiza exactamente $N-1$ intercambios (en el peor y mejor
caso, donde $N$ es el tamaño del arreglo).
Costo de Movimiento: Aunque el intercambio es costoso porque mueve el registro
completo, este costo solo se paga una vez
por pasada.
Costo de Comparación: El costo de las comparaciones O(N^2)S sigue siendo alto,
pero en escenarios donde
el movimiento es mucho más costoso que la comparación, Selection Sort es la
mejor opción cuadrática.*/
```

EJERCICIO 4

Libro.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>

struct Libro {
    std::string titulo;
    std::string autor;
    int anio;
    std::string ISBN;

    void print() const {
        std::cout << "[Autor: " << std::setw(15) << std::left << autor
                  << "| Año: " << std::setw(4) << anio
                  << "| Título: " << std::setw(25) << std::left << titulo <<
"]";
    }
};

struct Nodo {
    Libro data;
    Nodo* prev;
    Nodo* next;

    Nodo(const Libro& l) : data(l), prev(nullptr), next(nullptr) {}
};

int totalComparaciones = 0;
int totalMovimientosPunteros = 0;
int totalNodosIntercambiados = 0;
```

```
bool esMenor(const Libro& l1, const Libro& l2) {
    totalComparaciones++;

    if (l1.autor != l2.autor) {
        return l1.autor < l2.autor;
    }

    return l1.anio < l2.anio;
}

class ListaDoble {
private:
    Nodo* head;
    Nodo* tail;

    void swapNodes(Nodo* n1, Nodo* n2) {
        if (n1 == n2) return;

        totalMovimientosPunteros += 8;
        totalNodosIntercambiados++;

        Nodo* n1_prev = n1->prev;
        Nodo* n1_next = n1->next;

        Nodo* n2_prev = n2->prev;
        Nodo* n2_next = n2->next;

        if (n1_next == n2) {
            if (n1_prev) n1_prev->next = n2; else head = n2;
            n2->prev = n1_prev;

            n2->next = n1;
            n1->prev = n2;

            n1->next = n2_next;
            if (n2_next) n2_next->prev = n1; else tail = n1;
        }
        else {
            if (n1_prev) n1_prev->next = n2; else head = n2;
            if (n1_next) n1_next->prev = n2;

            if (n2_prev) n2_prev->next = n1; else head = n1;
            if (n2_next) n2_next->prev = n1;

            n1->next = n2_next;
            n1->prev = n2_prev;
            n2->next = n1_next;
            n2->prev = n1_prev;
        }
```

```
        if (!head) head = n1;
        while (tail && tail->next) tail = tail->next;

        std::cout << " \n";
    }

public:
    ListaDoble() : head(nullptr), tail(nullptr) {}
    ~ListaDoble() { }

    void add(const Libro& l) {
        Nodo* newNode = new Nodo(l);
        if (!head) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    void print() const {
        Nodo* current = head;
        if (!current) {
            std::cout << "Lista vacía.\n";
            return;
        }
        while (current != nullptr) {
            current->data.print();
            if (current->next) {
                std::cout << " <=> \n";
            }
            current = current->next;
        }
        std::cout << " <=> NULL\n";
    }

    void selectionSort() {
        if (!head || !head->next) return;

        Nodo* i = head;
        int pass = 1;

        std::cout <<
"\n============================================================\n";
        std::cout << "       INICIO DE ORDENAMIENTO (SELECTION SORT DOBLE)
\n";
        std::cout <<
"============================================================\n";

        while (i->next != nullptr) {
```

```
            Nodo* minNode = i;
            Nodo* j = i->next;

            std::cout << "\n--- Pasada " << pass << " (Buscando el mínimo
desde: ";
            i->data.print();
            std::cout << " ) ---\n";

            while (j != nullptr) {
                if (esMenor(j->data, minNode->data)) {
                    minNode = j;
                }
                j = j->next;
            }

            if (minNode != i) {
                std::cout << "  -> Mínimo encontrado: ";
                minNode->data.print();
                std::cout << "\n";

                swapNodes(i, minNode);

                i = minNode;
            } else {
                std::cout << "  -> Mínimo es el nodo actual. No hay
intercambio.\n";
            }

            i = i->next;
            pass++;

            std::cout <<
"------------------------------------------------------------\n";
        }
    }

    void showFinalAnalysis() const {
        std::cout <<
"\n============================================================\n";
        std::cout << "             ANÁLISIS FINAL DEL ALGORITMO
\n";
        std::cout <<
"============================================================\n";

        std::cout << "Registro de Operaciones:\n";
        std::cout << std::setw(35) << std::left << "Total de Comparaciones
realizadas:" << totalComparaciones << "\n";
        std::cout << std::setw(35) << std::left << "Total de Nodos
Intercambiados:" << totalNodosIntercambiados << "\n";
        std::cout << std::setw(35) << std::left << "Total de Movimientos de
Punteros:" << totalMovimientosPunteros << "\n";
```

```
    }

    void showResultsTable() const {
        std::cout << "\n--- Tabla de Resultados (Simulada) ---\n";
        std::cout << "+----------------------+--------------------
+----------------------+\n";
        std::cout << "| Intercambio #         | Comparaciones (Total)|
Movimientos Punteros |\n";
        std::cout << "+----------------------+--------------------
+----------------------+\n";
        std::cout << "| Total                 | " << std::setw(19) <<
totalComparaciones << " | " << std::setw(20) << totalMovimientosPunteros << " |
\n";
        std::cout << "+----------------------+--------------------
+----------------------+\n";
    }
};
```

main.cpp

```
#include <iostream>

#include <locale.h>
#include "Libro.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    ListaDoble libros;

    // Construir la lista con datos desordenados para probar la lógica de
comparación y el swap
    cout << "--- Construcción de la Lista Doblemente Enlazada ---\n";
    libros.add({"El Quijote", "Cervantes, M.", 1605, "1234"}); // A
    libros.add({"Cien Años", "García Márquez, G.", 1967, "5678"}); // B (Mínimo
por Autor)
    libros.add({"La Odisea", "Homero", -800, "9012"}); // C
    libros.add({"El Amor", "García Márquez, G.", 1985, "3456"}); // D (Mismo
Autor que B, Mayor Año)
    libros.add({"Fundacion", "Asimov, I.", 1951, "7890"}); // E

    cout << "Lista Original (Desordenada):\n";
    libros.print();

    // Aplicar Selection Sort
    libros.selectionSort();

    cout << "\n--- Lista Final Ordenada (Autor -> Año) ---\n";
```

```
    libros.print();

    // Presentar el análisis final y la tabla de resultados
    libros.showResultsTable();
    libros.showFinalAnalysis();

    return 0;
}

/*Ventajas de la Lista Doblemente Enlazada para Selection Sort:
Simplificación del Intercambio (Relativa): Aunque la lógica de intercambio de nodos sigue siendo compleja,
la presencia del puntero `prev` facilita re-enlazar los nodos vecinos.
En una lista simple, re-enlazar el nodo anterior requeriría recorrer toda la lista desde el inicio para encontrarlo.

Acceso Bidireccional (Potencial): Permite recorrer la lista hacia adelante (`next`) para la búsqueda del mínimo y,
si fuera necesario para una adaptación del algoritmo, retroceder (`prev`).
Manipulación de Punteros Fija: A pesar de la complejidad, el intercambio de dos nodos requiere un número fijo
(constante, O(1)) de re-asignaciones de punteros (generalmente 6-8 punteros) independientemente del tamaño de la lista,
lo cual es más eficiente que el desplazamiento de registros completos en un arreglo pesado.*/
```

EJERCICIO 5

ColaUsuario.h

```
#pragma once


#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
#include <cmath>
#include <algorithm>

struct Usuario {
    std::string id_usuario;
    int tiempo_espera;
};

struct NodoCola {
    Usuario user;
    NodoCola* next;
    NodoCola(const Usuario& u) : user(u), next(nullptr) {}
};

class ColaUsuarios {
```

```
private:
    NodoCola* frente;
    NodoCola* final;
    int numUsuarios;

public:
    ColaUsuarios() : frente(nullptr), final(nullptr), numUsuarios(0) {}

    int getNumUsuarios() const {
        return numUsuarios;
    }

    void enqueue(const Usuario& u) {
        NodoCola* nuevo = new NodoCola(u);
        if (final) {
            final->next = nuevo;
        } else {
            frente = nuevo;
        }
        final = nuevo;
        numUsuarios++;
    }

    double calcularPromedio() const {
        if (numUsuarios == 0) return 0.0;

        double sumaTiempos = 0.0;
        NodoCola* actual = frente;
        while (actual) {
            sumaTiempos += actual->user.tiempo_espera;
            actual = actual->next;
        }
        return std::round((sumaTiempos / numUsuarios) * 100.0) / 100.0;
    }

    void print() const {
        std::cout << "[Promedio: " << std::fixed << std::setprecision(2) <<
calcularPromedio() << " min, Usuarios: " << numUsuarios << "]";
    }
};

template <typename T>
void printArray(const std::vector<T>& arr) {
    std::cout << "+-------------------------------------------------+\n";
    std::cout << "| Posición | Contenido de la Cola                   |\n";
    std::cout << "+----------+----------------------------------------+\n";
    for (size_t i = 0; i < arr.size(); ++i) {
        std::cout << "| " << std::setw(8) << std::left << i << " | ";
        arr[i].print();
```

```
        std::cout << std::string(38 - 19 -
std::to_string(arr[i].calcularPromedio()).length() -
std::to_string(arr[i].getNumUsuarios()).length(), ' ') << "|\n";
    }
    std::cout << "+----------+---------------------------------------+\n";
}

template <typename T>
void bubbleSortOptimizado(std::vector<T>& arr) {
    int n = arr.size();
    bool intercambiado;
    int totalComparaciones = 0;
    int totalIntercambios = 0;
    std::vector<double> historialPromedios;

    std::cout <<
"\n=================================================================\n";
    std::cout << "       INICIO DE ORDENAMIENTO (BUBBLE SORT OPTIMIZADO)\n";
    std::cout <<
"=================================================================\n";

    for (int i = 0; i < n - 1; ++i) {
        intercambiado = false;
        int compPasada = 0;
        int intercPasada = 0;

        std::cout << "\n--- Pasada " << (i + 1) << " ---\n";

        for (int j = 0; j < n - 1 - i; ++j) {
            if (arr[j].calcularPromedio() > arr[j + 1].calcularPromedio()) {
                std::swap(arr[j], arr[j + 1]);
                intercambiado = true;
                totalIntercambios++;
                intercPasada++;
            }
            totalComparaciones++;
            compPasada++;
        }

        std::cout << "  -> Comparaciones realizadas: " << compPasada << "\n";
        std::cout << "  -> Intercambios efectuados: " << intercPasada << "\n";

        std::cout << "  -> Promedios después de la pasada: ";
        for(const auto& cola : arr) {
            double prom = cola.calcularPromedio();
            historialPromedios.push_back(prom);
            std::cout << prom << " | ";
        }
        std::cout << "\n";

        if (!intercambiado) {
```

```
            std::cout << "  -> **OPTIMIZACIÓN DETECTADA:** Pasada " << (i + 1)
<< " finalizada sin cambios. El arreglo está ordenado. Se detiene el algoritmo.
\n";
            break;
        }
    }

    // Análisis Final
    std::cout <<
"\n================================================================\n";
    std::cout << "                            ANÁLISIS FINAL\n";
    std::cout <<
"================================================================\n";
    std::cout << "Total de Comparaciones: " << totalComparaciones << "\n";
    std::cout << "Total de Intercambios: " << totalIntercambios << "\n";
}
```

main.cpp

```
#include <iostream>
#include <locale.h>
#include <vector>
#include "ColaUsuarios.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    std::vector<ColaUsuarios> arregloColas;

    ColaUsuarios c1;
    c1.enqueue({"U001", 10});
    c1.enqueue({"U002", 15});
    c1.enqueue({"U003", 15});
    arregloColas.push_back(c1);

    ColaUsuarios c2;
    c2.enqueue({"U010", 5});
    c2.enqueue({"U011", 5});
    arregloColas.push_back(c2);

    ColaUsuarios c3;
    c3.enqueue({"U020", 8});
    c3.enqueue({"U021", 12});
    c3.enqueue({"U022", 10});
    arregloColas.push_back(c3);

    ColaUsuarios c4;
    c4.enqueue({"U030", 11});
    arregloColas.push_back(c4);
```

```
    cout << "--- Estado Inicial del Arreglo de Colas ---\n";
    printArray(arregloColas);

    // Aplicar Bubble Sort Optimizado
    bubbleSortOptimizado(arregloColas);

    cout << "\n--- Estado Final del Arreglo (Ordenado por Promedio) ---\n";
    printArray(arregloColas);

    return 0;
}

/*Ineficiencia de Bubble Sort con Comparaciones Costosas:
Bubble Sort se vuelve ineficiente porque en cada comparación
(`arr[j].calcularPromedio() > arr[j+1].calcularPromedio()`), debe llamar a una
función
que requiere **recorrer y sumar** todos los elementos internos de la cola.

Si cada cola tiene K elementos, la comparación es O(K).
Como Bubble Sort tiene una complejidad O(N^2) en el peor caso,
la complejidad total se convierte en O(N^2 * K), donde:
N: Número de colas en el arreglo.
K: Número promedio de usuarios en cada cola.
Si K es grande, el factor O(N^2) se amplifica enormemente, haciendo el
algoritmo muy lento.*/
```

EJERCICIO 6

Tarea.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>
#include <vector>

struct Tarea
{
    int prioridad;
    std::string descripcion;

    // Constructor simple para la lectura de la proridad y descripcopn / si hay
tiempo evitamos usarlo
    Tarea(int p, const std::string &d) : prioridad(p), descripcion(d) {}

    void print() const
    {
        std::cout << "[P" << prioridad << " - " << descripcion << "]";
    }
};
```

```
struct NodoTarea
{
    Tarea data;

    NodoTarea(int p, const std::string &d) : data(p, d) {}
};

template <typename T>
void printVector(const std::vector<T *> &arr)
{
    std::cout << "Índice (Vector): |";
    for (size_t i = 0; i < arr.size(); ++i)
    {
        std::cout << std::setw(9) << std::right << i << " |";
    }
    std::cout << "\n------------------";
    for (size_t i = 0; i < arr.size(); ++i)
    {
        std::cout << "---------------";
    }
    std::cout << "\nValor Apuntado:    |";
    for (const auto &ptr : arr)
    {

        std::cout << std::setw(8) << std::right << "P" << ptr->data.prioridad
<< " |";
    }
    std::cout << "\n";
}

template <typename T>
void bubbleSortPunteros(std::vector<T *> &ptr_arr)
{
    int n = ptr_arr.size();
    bool intercambiado;
    int totalComparaciones = 0;
    int totalIntercambios = 0;

    std::cout <<
"\n======================================================================\n";
    std::cout << "  INICIO DE ORDENAMIENTO (BUBBLE SORT INTERCAMBIO DE
PUNTEROS)       \n";
    std::cout <<
"======================================================================\n";

    for (int i = 0; i < n - 1; ++i)
    {
        intercambiado = false;
        int compPasada = 0;
        int intercPasada = 0;
```

```
        std::cout << "\n--- Pasada " << (i + 1) << " ---\n";

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

            if (ptr_arr[j]->data.prioridad > ptr_arr[j + 1]->data.prioridad)
            {

                std::swap(ptr_arr[j], ptr_arr[j + 1]);

                intercambiado = true;
                totalIntercambios++;
                intercPasada++;
            }
            totalComparaciones++;
            compPasada++;
        }

        std::cout << "  -> Comparaciones en esta pasada: " << compPasada <<
"\n";
        std::cout << "  -> Intercambios en esta pasada: " << intercPasada <<
"\n";

        std::cout << "\nEstado del Vector de Punteros tras la Pasada " << (i +
1) << ":\n";
        printVector(ptr_arr);

        if (!intercambiado)
        {
            std::cout << "  -> **OPTIMIZACIÓN:** Arreglo ordenado. Deteniendo.
\n";
            break;
        }
    }

    std::cout <<
"\n================================================================\n";
    std::cout << "                          ANÁLISIS FINAL\n";
    std::cout <<
"================================================================\n";
    std::cout << "Total de Comparaciones: " << totalComparaciones << "\n";
    std::cout << "Total de Intercambios de Punteros: " << totalIntercambios <<
"\n";
}
```

main.cpp

```
#include <iostream>
#include <locale.h>
#include <vector>
```

```
#include "Tarea.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    // 1. Crear los nodos dinámicos (registros pesados)
    // Se crean y permanecen en memoria, solo movemos las referencias.
    NodoTarea* n1 = new NodoTarea(5, "Revisar Documentos"); // Menor prioridad
(debe ir al final)
    NodoTarea* n2 = new NodoTarea(2, "Merge Request Urgente"); // Mayor
prioridad (debe ir al inicio)
    NodoTarea* n3 = new NodoTarea(4, "Reunión Equipo");
    NodoTarea* n4 = new NodoTarea(1, "Fix de Producción"); // Máxima prioridad

    // 2. Un vector contiene punteros a nodos que representan tareas.
    std::vector<NodoTarea*> vectorPunteros = {n1, n2, n3, n4};

    cout << "--- Estado Inicial del Vector de Punteros (Orden por Prioridad)
---\n";
    printVector(vectorPunteros);

    // Aplicar Bubble Sort Intercambiando Punteros
    bubbleSortPunteros(vectorPunteros);

    cout << "\n--- Estado Final del Vector (Ordenado por Prioridad) ---\n";
    printVector(vectorPunteros);

    // IMPORTANTE: Liberación de memoria dinámica
    for (NodoTarea* ptr : vectorPunteros) {
        delete ptr;
    }

    return 0;
}

/*Beneficio de Mover Punteros en Lugar de Registros Completos:
1.Costo de Movimiento:Mover un registro grande es costoso (implica muchas
copias de memoria).
Mover un puntero es extremadamente rápido O(1)
2.Eficiencia del Intercambio:Al usar `std::swap(puntero1, puntero2)`, solo se
intercambian
las direcciones en el vector, mientras que la estructura de datos `Tarea`
permanece estática
en la memoria.
El costo de Bubble Sort sigue siendo O(N^2) en comparaciones, pero el costo
constante por cada intercambio es mínimo.
3.Aplicación:Esta técnica es esencial cuando se ordenan estructuras grandes en
memoria o cuando se requiere que los
datos originales no se muevan de su ubicación física .*/
```

EJERCICIO 7

Persona.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>
#include <vector>

struct Direccion {
    std::string calle;
    int numero;
    std::string ciudad;
};

struct Persona {
    int id;
    std::string nombre;
    int edad;
    Direccion direccion;

    void print() const {
        std::cout << "|" << std::setw(3) << id
                  << "|" << std::setw(15) << std::left << nombre
                  << "|" << std::setw(5) << edad
                  << "|" << std::setw(15) << std::left << direccion.ciudad
                  << "|" << std::setw(15) << std::left << direccion.calle <<
"|";
    }
};

int totalComparaciones = 0;
int totalIntercambios = 0;

bool esMenor(const Persona& p1, const Persona& p2) {
    totalComparaciones++;

    if (p1.direccion.ciudad != p2.direccion.ciudad) {
        return p1.direccion.ciudad < p2.direccion.ciudad;
    }

    totalComparaciones++;
    // 2. Criterio: Edad (Ascendente)
    if (p1.edad != p2.edad) {
        return p1.edad < p2.edad;
    }

    totalComparaciones++;
```

```
    return p1.nombre < p2.nombre;
}

template <typename T>
void exchangeSort(std::vector<T>& arr) {
    int n = arr.size();

    std::cout <<
"\n========================================================================\n";
    std::cout << "      INICIO DE ORDENAMIENTO (EXCHANGE SORT CON CRITERIO
COMPLEJO)    \n";
    std::cout <<
"========================================================================\n";

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

            if (esMenor(arr[j], arr[i])) {

                std::cout << "\n  -> Intercambio forzado:\n";
                std::cout << "     - Elemento i: "; arr[i].print(); std::cout
<< "\n";
                std::cout << "     - Elemento j: "; arr[j].print(); std::cout
<< "\n";

                std::swap(arr[i], arr[j]);
                totalIntercambios++;
            }
        }

        std::cout << "\nEstado del Arreglo tras la Pasada Externa " << (i + 1)
<< ":\n";
        std::cout << "+---+---------------+-----+--------------
+---------------+\n";
        std::cout << "| ID| Nombre        | Edad| Ciudad        | Calle
|\n";
        std::cout << "+---+---------------+-----+--------------
+---------------+\n";
        for (const auto& p : arr) {
            p.print();
            std::cout << "\n";
        }
        std::cout << "+---+---------------+-----+--------------
+---------------+\n";
    }

    std::cout <<
"\n========================================================================\n";
    std::cout << "                          ANÁLISIS FINAL\n";
    std::cout <<
"========================================================================\n";
```

```
    std::cout << "Total de Comparaciones (Aprox. N^2 * 3 niveles): " <<
totalComparaciones << "\n";
    std::cout << "Total de Intercambios realizados: " << totalIntercambios <<
"\n";


}
```

main.cpp

```
#include <iostream>
#include <locale.h>
#include <vector>
#include "Persona.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    std::vector<Persona> personas = {

        {101, "Carlos", 35, {"Principal", 101, "Quito"}},
        {204, "Ana",    28, {"Calle A", 50,  "Guayaquil"}},
        {310, "Elena",  35, {"Central", 200, "Quito"}},
        {402, "Bruno",  40, {"Av. Sur", 15,  "Guayaquil"}},
        {507, "Daniel", 28, {"2da Av.", 70,  "Guayaquil"}}
    };

    cout << "--- Estado Inicial del Arreglo de Personas ---\n";
    cout << "+---+---------------+-----+---------------+---------------+\n";
    cout << "| ID| Nombre        | Edad| Ciudad        | Calle         |\n";
    cout << "+---+---------------+-----+---------------+---------------+\n";
    for (const auto& p : personas) {
        p.print();
        cout << "\n";
    }
    cout << "+---+---------------+-----+---------------+---------------+\n";

    exchangeSort(personas);

    cout << "\n--- Estado Final del Arreglo (Ordenado: Ciudad -> Edad ->
Nombre) ---\n";
    cout << "+---+---------------+-----+---------------+---------------+\n";
    cout << "| ID| Nombre        | Edad| Ciudad        | Calle         |\n";
    cout << "+---+---------------+-----+---------------+---------------+\n";
    for (const auto& p : personas) {
        p.print();
        cout << "\n";
    }
    cout << "+---+---------------+-----+---------------+---------------+\n";
```

```
    return 0;
}

/*Impacto de Exchange Sort con Registros Complejos y Pesados:
1.Costo de Comparación (Costo Fijo Alto):Cada comparación requiere evaluar tres
niveles(`ciudad`, `edad`, `nombre`).
Esto hace que el costo constante de la operación O(1) sea alto. Dado que el
algoritmo es O(n^2) en comparaciones,
este alto costo se multiplica por N^2.

2.Costo de Movimiento (Costo Variable Alto):Cada intercambio (`std::swap`)
implica copiar la estructura completa,
incluyendo la sub-estructura `Direccion`. Si la estructura `Persona` es
'pesada', el costo de mover estos datos
es alto y se ejecuta tantas veces como sea necesario (puede ser hasta
$athcal{O}(N^2)$ intercambios).

3.Costo en Memoria por Mover Registros Grandes:El costo en memoria
(tiempo/latencia) de mover registros grandes
es significativo. En Exchange Sort, el número de intercambios no es fijo como
en Selection Sort, lo que significa
que en el peor caso, Exchange Sort es menos eficiente para estructuras pesadas
que Selection Sort,
debido al potencial de muchos más intercambios.\n";*/
```

EJERCICIO 8

Venta.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>
#include <vector>

struct Venta {
    std::string id_venta;
    std::string fecha;
    double valor;

    int order;

    void print() const {
        std::cout << "|" << std::setw(3) << order
                  << "|" << std::setw(10) << std::left << id_venta
                  << "|" << std::setw(10) << std::left << fecha
                  << "|" << std::setw(8) << std::right << std::fixed <<
std::setprecision(2) << valor << "|";
    }
```

```
};

template <typename T>
void printArray(const std::vector<T>& arr) {
    std::cout << "+---+----------+----------+--------+\n";
    std::cout << "|Ord| ID Venta | Fecha    | Valor  |\n";
    std::cout << "+---+----------+----------+--------+\n";
    for (const auto& item : arr) {
        item.print();
        std::cout << "\n";
    }
    std::cout << "+---+----------+----------+--------+\n";
}

int totalComparaciones = 0;
int totalIntercambios = 0;

template <typename T>
void exchangeSort(std::vector<T>& arr) {
    int n = arr.size();

    std::cout <<
"\n=========================================================\n";
    std::cout << "        INICIO DE ORDENAMIENTO (EXCHANGE SORT)\n";
    std::cout << "=========================================================\n";

    for (int i = 0; i < n - 1; ++i) {
        std::cout << "\n--- Pasada Externa i = " << i << " ---\n";

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

            totalComparaciones++;
            std::cout << "  -> Comparando Valor[" << i << "] (" << arr[i].valor
<< ") vs Valor[" << j << "] (" << arr[j].valor << ").\n";

            if (arr[j].valor < arr[i].valor) {

                std::cout << "  -> INTERCAMBIO: Los elementos están fuera de
orden. Swapping Venta(Ord " << arr[i].order << ") con Venta(Ord " <<
arr[j].order << ").\n";

                std::swap(arr[i], arr[j]);
                totalIntercambios++;
            }
        }

        std::cout << "\nEstado del Arreglo después de la Pasada " << (i + 1) <<
":\n";
        printArray(arr);
    }
}
```

```
template <typename T>
void analizarEstabilidad(const std::vector<T>& arr) {
    std::cout <<
"\n=======================================================\n";
    std::cout << "           ANÁLISIS FORMAL DE ESTABILIDAD\n";
    std::cout << "=======================================================\n";

    int index2 = -1;
    int index3 = -1;

    for (size_t i = 0; i < arr.size(); ++i) {
        if (arr[i].order == 2) index2 = i;
        if (arr[i].order == 3) index3 = i;
    }

    std::cout << "Elementos de prueba (Valor = $200.00):\n";
    std::cout << "  - V102 (Orden Original 2) está en la posición final: " <<
index2 << "\n";
    std::cout << "  - V103 (Orden Original 3) está en la posición final: " <<
index3 << "\n";

    std::cout << "\nComparación de Índices:\n";
    if (index2 < index3) {
        std::cout << "V102 (Ord 2) < V103 (Ord 3): El orden original se ha
preservado.\n";
    } else {
        std::cout << "V102 (Ord 2) > V103 (Ord 3): El orden original NO se ha
preservado.\n";
    }


    std::cout << "\n--- Resumen de Costos ---\n";
    std::cout << "Total de Comparaciones: " << totalComparaciones << "\n";
    std::cout << "Total de Intercambios: " << totalIntercambios << "\n";
}
```

main.cpp

```
#include <iostream>
#include <locale.h>
#include <vector>
#include "Venta.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    std::vector<Venta> ventas = {
        {"V101", "2025-01-10", 350.00, 1},
```

```
        {"V102", "2025-01-11", 200.00, 2},
        {"V103", "2025-01-12", 200.00, 3},
        {"V104", "2025-01-09", 500.00, 4},
        {"V105", "2025-01-13", 150.00, 5}
    };

    cout << "--- Estado Inicial del Arreglo de Ventas ---\n";
    printArray(ventas);


    exchangeSort(ventas);

    cout << "\n--- Estado Final del Arreglo (Ordenado por Valor) ---\n";
    printArray(ventas);

    analizarEstabilidad(ventas);

    return 0;
}
```

EJERCICIO 9

ProductoArbol.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>

struct Producto
{
    std::string codigo;
    std::string nombre;
    double precio;
    std::string categoria;

    void print() const
    {
        std::cout << "|" << std::setw(8) << std::left << codigo
                  << "|" << std::setw(15) << std::left << nombre
                  << "|" << std::setw(8) << std::right << std::fixed <<
std::setprecision(2) << precio
                  << "|" << std::setw(12) << std::left << categoria << "|";
```

```
    }
};

template <typename T>
void printSubArray(const std::vector<T> &arr, int low, int high)
{
    std::cout << "[";
    for (int i = low; i <= high; ++i)
    {
        std::cout << arr[i].nombre;
        if (i < high)
        {
            std::cout << ", ";
        }
    }
    std::cout << "]";
}

std::vector<std::string> recursionTree;

template <typename T>
int partition(std::vector<T> &arr, int low, int high, int &movimientos)
{

    std::srand(std::time(0));
    int random_idx = low + (std::rand() % (high - low + 1));
    std::swap(arr[random_idx], arr[high]);

    T pivot = arr[high];
    int i = low - 1;
    int movPasada = 0;

    std::cout << "  -> Pivote seleccionado (Aleatorio) en índice " << high <<
": " << pivot.nombre << " (Precio: " << pivot.precio << ")\n";
    std::cout << "  -> Índices usados (low=" << low << ", high=" << high << ")
\n";


    for (int j = low; j < high; ++j)
    {
        if (arr[j].precio <= pivot.precio)
        {
            i++;
            std::swap(arr[i], arr[j]);
            movimientos++;
            movPasada++;
        }
    }

    std::swap(arr[i + 1], arr[high]);
    movimientos++;
```

```
    movPasada++;

    std::cout << "  -> Movimientos de elementos realizados en la partición: "
<< movPasada << "\n";

    return i + 1;
}

template <typename T>
void quickSort(std::vector<T> &arr, int low, int high, int &movimientos, int
level = 0)
{
    if (low < high)
    {

        std::string trace = "| Nivel " + std::to_string(level) + " |
Subarreglo: ";
        trace += "[" + std::to_string(low) + "-" + std::to_string(high) + "] ";

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

        trace += "-> Pivote final: " + arr[pi].nombre;
        trace += " -> Subarreglos: Izq [" + std::to_string(low) + "-" +
std::to_string(pi - 1) + "]";
        trace += ", Der [" + std::to_string(pi + 1) + "-" +
std::to_string(high) + "]";
        recursionTree.push_back(trace);

        quickSort(arr, low, pi - 1, movimientos, level + 1);
        quickSort(arr, pi + 1, high, movimientos, level + 1);
    }
}

template <typename T>
void sortArray(std::vector<T> &arr)
{
    int movimientos = 0;
    std::srand(static_cast<unsigned int>(std::time(0)));

    std::cout <<
"\n=======================================================\n";
    std::cout << "       INICIO DE ORDENAMIENTO (QUICKSORT ALEATORIO)\n";
    std::cout << "=======================================================\n";

    quickSort(arr, 0, arr.size() - 1, movimientos);

    std::cout <<
"\n=======================================================\n";
    std::cout << "             ANÁLISIS FINAL DE QUICKSORT\n";
    std::cout << "=======================================================\n";
```

```
    std::cout << "Total de Movimientos de Elementos (Swaps en Partición): " <<
movimientos << "\n";

    std::cout << "\nÁrbol de Recursión Generado:\n";
    for (const auto &line : recursionTree)
    {
        std::cout << line << "\n";
    }
}
```

main.cpp

```
// main.cpp

#include <iostream>
#include <locale.h>
#include <vector>
#include "ProductoArbol.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    std::vector<Producto> productos = {
        // Código | Nombre          | Precio | Categoría
        {"D01", "Destornillador", 15.00, "Herramienta"}, // P0
        {"M05", "Martillo",       30.00, "Herramienta"}, // P1
        {"T03", "Tuerca M5",       1.50, "Fijacion"},    // P2
        {"B02", "Broca 1/4",      12.00, "Herramienta"}, // P3
        {"C01", "Cinta Metrica",  20.00, "Medicion"},    // P4
        {"C02", "Clavos",          5.00, "Fijacion"}     // P5
    };

    cout << "--- Estado Inicial del Vector de Productos ---\n";
    cout << "+--------+---------------+--------+------------+\n";
    cout << "| Código | Nombre        | Precio | Categoría  |\n";
    cout << "+--------+---------------+--------+------------+\n";
    for (const auto& p : productos) {
        p.print();
        cout << "\n";
    }
    cout << "+--------+---------------+--------+------------+\n";

    sortArray(productos);

    cout << "\n--- Estado Final del Vector (Ordenado por Precio) ---\n";
    cout << "+--------+---------------+--------+------------+\n";
    cout << "| Código | Nombre        | Precio | Categoría  |\n";
    cout << "+--------+---------------+--------+------------+\n";
    for (const auto& p : productos) {
        p.print();
```

```
        cout << "\n";
    }
    cout << "+--------+---------------+--------+------------+\n";

    return 0;
}

/*Análisis de Rendimiento y Aleatorización del Pivote:

1. Rendimiento Promedio: Quicksort tiene un rendimiento promedio excelente de
O(N log N).

2. Impacto de la Aleatorización: La aleatorización del pivote es una estrategia
para evitar el peor caso O(N^2), que ocurre cuando el pivote siempre resulta
ser
el elemento más pequeño o el más grande.

3. Garantía Probabilística: Al elegir un pivote aleatorio, se asegura que,
en promedio, el arreglo se dividirá en dos subarreglos de tamaño similar,
manteniendo la complejidad en O(N log N) con alta probabilidad,
independientemente del orden inicial de los datos.*/
```

EJERCICIO 10

Proceso.h

```
#pragma once


#include <iostream>
#include <string>
#include <iomanip>
#include <algorithm>

using namespace std;

struct Proceso {
    int pid;
    int tiempo_cpu;
    int prioridad;

    void print() const {
        std::cout << "[PID: " << pid << ", Prio: " << prioridad << ", CPU: " <<
tiempo_cpu << "ms]";
    }
```

```
};

struct Nodo {
    Proceso data;
    Nodo* prev;
    Nodo* next;

    Nodo(const Proceso& p) : data(p), prev(nullptr), next(nullptr) {}
};

class ListaDobleCircular {
private:
    Nodo* head;

    void printRecursive(Nodo* start) const {
        if (!start) {
             std::cout << "Lista vacía.\n";
             return;
        }
        Nodo* current = start;
        do {
            current->data.print();
            if (current->next != start) {
                std::cout << " <=> \n";
            }
            current = current->next;
        } while (current != start);
        std::cout << " <=> (HEAD)\n";
    }

    Nodo* partition(Nodo* low, Nodo* high) {
        if (!low || !high || low == high || low->prev == high) return low;

        int pivot_prioridad = high->data.prioridad;

        Nodo* i_before_low = low->prev;
        Nodo* i = i_before_low;
        Nodo* j = low;

        std::cout << "\n  -> Pivote: "; high->data.print();
        std::cout << "\n  -> Rango: ["; low->data.print(); std::cout << " ...
"; high->data.print(); std::cout << "]\n";

        while (j != high) {
            if (j->data.prioridad <= pivot_prioridad) {
                i = (i == i_before_low) ? low : i->next;

                if (i != j) {
                    std::swap(i->data, j->data);
                    std::cout << "    * Swap de datos: i (P" << i-
>data.prioridad << ") <-> j (P" << j->data.prioridad << ")\n";
```

```
                }
            }
            j = j->next;
        }

        i = (i == i_before_low) ? low : i->next;

        std::swap(i->data, high->data);
        std::cout << "  -> Pivote final colocado en: "; i->data.print();
std::cout << "\n";

        return i;
    }

    void quickSortRecursive(Nodo* low, Nodo* high, int level) {
        if (!low || !high || low == high) return;
        if (low->next == high && low->data.prioridad <= high->data.prioridad)
return;

        std::cout << "\n--- Llamada Recursiva (Nivel " << level << ") ---\n";

        Nodo* pivot = partition(low, high);

        if (pivot != low && pivot->prev != low) {
             quickSortRecursive(low, pivot->prev, level + 1);
        }

        if (pivot != high && pivot->next != high) {
            quickSortRecursive(pivot->next, high, level + 1);
        }

        std::cout << "\n[Lista tras Nivel " << level << "]: ";
        printRecursive(head);
    }

public:
    ListaDobleCircular() : head(nullptr) {}

    void add(const Proceso& p) {
        Nodo* nuevo = new Nodo(p);
        if (!head) {
            head = nuevo;
            head->next = head;
            head->prev = head;
        } else {
            Nodo* last = head->prev;
            nuevo->next = head;
            nuevo->prev = last;
            head->prev = nuevo;
            last->next = nuevo;
        }
```

```
    }

    void print() const {
        printRecursive(head);
    }

    void sort() {
        if (!head || head->next == head) return;

        Nodo* tail = head->prev;

        quickSortRecursive(head, tail, 0);
    }

    void showAnalysis() const {
        std::cout <<
"\n=======================================================\n";
        std::cout << "             ANÁLISIS DE QUICKSORT EN D-L-C\n";
        std::cout <<
"=======================================================\n";

        std::cout << "\nDesafíos de aplicar Quicksort a Estructuras Enlazadas:
\n";
        std::cout << "1. Sin Acceso Aleatorio O(N): Quicksort se basa en el
acceso por índice O(1) para las particiones. En listas enlazadas, el acceso a
un nodo k requiere O(k) pasos (recorrido secuencial).\n";
        std::cout << "2. Complejidad de la Partición: La partición requiere que
los elementos menores que el pivote se muevan a la izquierda y los mayores a la
derecha. Mover nodos completos en una lista doblemente enlazada circular
requiere manipular 4 punteros (`prev` y `next` de los nodos vecinos) por cada
nodo movido, haciendo que la partición sea costosa y propensa a errores.\n";
        std::cout << "3. Fusión de Sublistas: Una vez que la recursión devuelve
las sublistas ordenadas, es necesario re-conectar (`low->prev` y `high->next`)
los límites de las sublistas y ajustar la circularidad, lo cual añade otra capa
de dificultad.\n";

        std::cout << "\nComparación de Eficiencia con la Versión para Arreglos:
\n";
        std::cout << "La eficiencia teórica de Quicksort en listas enlazadas es
mucho menor.\n";
        std::cout << "Arreglos: O(N log N) en tiempo promedio (por
comparaciones) + costo constante de swaps.\n";
        std::cout << "Listas Enlazadas: La fase de partición (mover elementos
menores/mayores) es ineficiente. El costo de recorrer para encontrar límites o
elementos es alto. En la práctica, Quicksort en listas doblemente enlazadas
suele ser más lento que algoritmos simples como Merge Sort (que es ideal para
listas, ya que las divisiones son O(1) o Insertion Sort (para listas
pequeñas/casi ordenadas).\n";
    }
};
```

main.cpp

```
#include <iostream>

#include <locale.h>
#include "Proceso.h"

using namespace std;

int main() {
    setlocale(LC_CTYPE, "Spanish");

    ListaDobleCircular procesos;

    cout << "--- Construcción de la Lista Doblemente Enlazada Circular ---\n";
    procesos.add({101, 50, 4});
    procesos.add({204, 10, 2});
    procesos.add({310, 80, 5});
    procesos.add({402, 25, 3});
    procesos.add({507, 45, 1});

    cout << "Lista Original (Desordenada):\n";
    procesos.print();

    procesos.sort();

    cout << "\n--- Lista Final (Ordenada por Prioridad - Simulación) ---\n";
    procesos.print();

    // Presentar el análisis
    procesos.showAnalysis();

    return 0;
}
```