# Actividades de la clase 1

## Actividad 1.1:

1. MaximoArray tiene una complejidad asintótica de O(n) siendo n el tamaño del array.

public class MaximoArray {

public static void main(String[] args) {

int[] numeros = {1, 5, 3, 10, 5, 3};

int maximo = EncontrarMaximo(numeros);

System.out.println("El máximo valor en el array es: " + maximo);

}

public static int EncontrarMaximo(int[] nums) {

int max = nums [0];

for (int i = 1; i < nums.length; i++) {

if (nums[i] > max) {

max = nums[i];

}

}

return max;

}

}

1. Usar **Map** es O(n) porque recorre solo una vez las n facturas. Sin **Map**, la búsqueda en una lista de resultados de tamaño m requiere O(m) por factura, lo que resulta en O(n2) al procesar n facturas, ya que m puede crecer hasta n.

* Sin map:

import java.util.ArrayList;

import java.util.List;

public class FacturacionSinMap {

public static void main(String[] args) {

// Datos de entrada: Facturas

List<List<Object>> facturas = new ArrayList<>();

facturas.add(new ArrayList<>(List.of(1, 101, 500)));

facturas.add(new ArrayList<>(List.of(2, 102, 300)));

facturas.add(new ArrayList<>(List.of(3, 101, 200)));

facturas.add(new ArrayList<>(List.of(4, 103, 600)));

facturas.add(new ArrayList<>(List.of(5, 102, 150)));

// Datos de entrada: Clientes

List<List<Object>> clientes = new ArrayList<>();

clientes.add(new ArrayList<>(List.of(101, "Cliente A")));

clientes.add(new ArrayList<>(List.of(102, "Cliente B")));

clientes.add(new ArrayList<>(List.of(103, "Cliente C")));

// Lista para almacenar los resultados

List<List<Object>> resultados = new ArrayList<>();

// Recorrer la lista de facturas para acumular los importes

for (List<Object> factura : facturas) {

int idCliente = (int) factura.get(1);

int importe = (int) factura.get(2);

// Buscar si ya existe este cliente en la lista de resultados

int indiceCliente = buscarClienteEnResultados(resultados, idCliente);

if (indiceCliente != -1) {

List<Object> resultadoExistente = resultados.get(indiceCliente);

resultadoExistente.set(2, (int) resultadoExistente.get(2) + importe);

} else {

String nombreCliente = getNombreCliente(clientes, idCliente);

resultados.add(new ArrayList<>(List.of(idCliente, nombreCliente, importe)));

}

}

// Mostrar los resultados

for (List<Object> resultado : resultados) {

System.out.println("ID Cliente: " + resultado.get(0) + ", Nombre: " + resultado.get(1) + ", Suma Importes: " + resultado.get(2));

}

}

// Método auxiliar para buscar si un cliente ya está en los resultados

public static int buscarClienteEnResultados(List<List<Object>> resultados, int idCliente) {

for (int i = 0; i < resultados.size(); i++) {

if ((int) resultados.get(i).get(0) == idCliente) {

return i;

}

}

return -1;

}

// Método auxiliar para obtener el nombre del cliente por su ID

public static String getNombreCliente(List<List<Object>> clientes, int idCliente) {

for (List<Object> cliente : clientes) {

if ((int) cliente.get(0) == idCliente) {

return (String) cliente.get(1);

}

}

return null;

}

}

* Con map:

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class FacturacionConMap {

public static void main(String[] args) {

// Datos de entrada: Facturas

List<List<Object>> facturas = new ArrayList<>();

facturas.add(new ArrayList<>(List.of(1, 101, 500)));

facturas.add(new ArrayList<>(List.of(2, 102, 300)));

facturas.add(new ArrayList<>(List.of(3, 101, 200)));

facturas.add(new ArrayList<>(List.of(4, 103, 600)));

facturas.add(new ArrayList<>(List.of(5, 102, 150)));

// Datos de entrada: Clientes

List<List<Object>> clientes = new ArrayList<>();

clientes.add(new ArrayList<>(List.of(101, "Cliente A")));

clientes.add(new ArrayList<>(List.of(102, "Cliente B")));

clientes.add(new ArrayList<>(List.of(103, "Cliente C")));

// Crear un Map para almacenar la suma de importes por cliente

Map<Integer, Integer> sumaImportesPorCliente = new HashMap<>();

// Sumar los importes de las facturas por cliente

for (List<Object> factura : facturas) {

int idCliente = (int) factura.get(1);

int importe = (int) factura.get(2);

// Comprobar si el cliente ya está en el Map

if (sumaImportesPorCliente.containsKey(idCliente)) {

int sumaActual = sumaImportesPorCliente.get(idCliente);

sumaImportesPorCliente.put(idCliente, sumaActual + importe);

} else {

sumaImportesPorCliente.put(idCliente, importe);

}

}

// Lista para almacenar los resultados

List<List<Object>> resultados = new ArrayList<>();

// Crear la lista de resultados combinando clientes con suma de importes

for (List<Object> cliente : clientes) {

int idCliente = (int) cliente.get(0);

String nombreCliente = (String) cliente.get(1);

int sumaImportes = sumaImportesPorCliente.get(idCliente);

resultados.add(new ArrayList<>(List.of(idCliente, nombreCliente, sumaImportes)));

}

// Mostrar los resultados

for (List<Object> resultado : resultados) {

System.out.println("ID Cliente: " + resultado.get(0) + ", Nombre: " + resultado.get(1) + ", Suma Importes: " + resultado.get(2));

}

}

}

## Actividad 1.2:

* **long**: Es eficiente y rápido para operaciones aritméticas cuando los números están dentro del rango (-2^63 a 2^63-1). Se usa cuando sabes que los números no van a exceder este rango.

public class Long {

public static void main(String[] args) {

long num = 9223372036854775807L; // Max valor que puede almacenar un long

long sum = num + 1;

System.out.println("Numero grande (long): " + num);

System.out.println("Suma con desbordamiento: " + sum);

}

}

* **BigInteger**: Se utiliza para manejar números enteros que pueden ser grandes, como cuando trabajás con valores mucho mayores que los soportados por **long**. Las operaciones con **BigInteger** son más lentas que con **long** por la complejidad adicional de manejar números más grandes.

import java.math.BigInteger;

public class ClaseBigInteger {

public static void main(String[] args) {

BigInteger num1 = new BigInteger("9223372036854775807"); // Max valor de long

BigInteger num2 = new BigInteger("1000000000000000000000"); // Mayor al max

BigInteger sum = num1.add(num1);

BigInteger mult = num1.multiply(num1);

System.out.println("Numero muy grande (BigInteger): " + num1);

System.out.println("Otro numero grande (BigInteger): " + num2);

System.out.println("sum: " + sum);

System.out.println("mult: " + mult);

}

}

## Actividad 1.3:

* T(n) es el tiempo que toma sumar los primeros n números enteros.
* T(n−1) es el tiempo que toma sumar los primeros n-1 números enteros.
* O(1) representa el tiempo constante que toma realizar la operación de suma y la llamada

T(n) = T(n−1) + O(1)

= (T(n−2) + O(1)) + O(1)

= ((T(n−3) + O(1)) + O(1)) + O(1)

…

= T(0) + n × O(1)

Dado que T(0) es constante, podemos decir que: T(n)=O(n)

public class SumarNumerosRecursivamente {

public static int SumarNumeros(int n) {

if (n <= 0) {

return 0; // Caso base

} else {

return n + SumarNumeros(n - 1); // Llamada recursiva

}

}

public static void main(String[] args) {

int n = 10;

int resultado = SumarNumeros(n);

System.out.println("La suma de los primeros " + n + " numeros enteros es: " + resultado);

}  
}