**Guía de estudio**

Rodrigo Legarreta Soto

**¿Qué es la notación big O y para qué sirve?**

La notación big o es la complejidad de un Código, esta sirve como un análisis al Código, se basa en el tiempo de ejecución y en el uso de memoria, sirve mucho para comparar códigos de una forma objetiva, esto es importante para ver la eficiencia del Código, esta va midiendo como crece el tiempo de ejecución a medida que aumenta el tamaño de la entrada (**n**).

**¿Qué significa que es asintótico?**

la notación big O usa un algoritmo el cual es muy grande, por que es lo que nos interesa saber, en cuanto tiempo se ejecuta un programa tan grande, por lo tanto un comportamiento asintótico significa que nos interesa ver ese comportamiento a gran escala para poder ver la eficiencia de un programa, cuando n es igual a infinito es un comportamiento asintótico y es por eso que se usa para saber la complejidad del Código.

**Colocar grafica de comportamiento de big O**

Gráfico

El contenido generado por IA puede ser incorrecto.

**Identificar Algoritmo según su descripción**

**- Inserción**

El código de inserción es un método de ordenamiento de datos que es eficiente para arreglos pequeños, el funcionamiento de este se basa en asumir que el primer dato ya esta ordenado y a partir de ahí comparar el resto de la lista “ya ordenada” previamente, se elige un número y a partir de ese se compara hacia atrás.

**Peor caso O(n²)**

**Mejor Caso O(n)**

Este usa key

**- Selección**

En este código se busca el elemento más pequeño del arreglo y se manda hasta el inicio, se hace esto hasta que el arreglo queda ordenado.

**Peor caso O(n²)**

**Mejor caso O(n²)**

Este usa mini index

**- Bubble Sort**

Este código principalmente va comparando en parejas y esto lo hace con cada uno de los elementos hasta que queda ordenado, si están en el orden incorrecto después de la primera pasada se repite el proceso varias veces.

**Peor caso O(n²):**

**Mejor caso O(n):**

Este usa swapped true

**- Merge Sort**

Merge Sort es un algoritmo de ordenamiento que consiste en dividir un array en dos mitades, ordena cada mitad y luego fusiona las mitades ordenadas. Este proceso se repite hasta que todo el array esté ordenado.

El algoritmo sigue el enfoque de divide-and-conquer

Usa left right y middle

**- Quick Sort**

QuickSort es un algoritmo de ordenamiento eficiente. Funciona dividiendo un arreglo en subarreglos más pequeños, ordenándolos y combinándolos para obtener el resultado final, usa un pivote y a partir de ahí va dividiendo el arreglo y compara todo si es mayor o menor.

Este usa un pivote

**- búsqueda secuencial**

La búsqueda secuencial es un algoritmo utilizado para encontrar un elemento en una lista recorriendo uno por uno los elementos desde el inicio hasta el final. Se compara cada elemento con el valor buscado hasta encontrarlo o llegar al final de la lista.

Este usa un objetivo o target y se diferencia porque es muy pequeño

**- búsqueda Binaria**

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada. Funciona dividiendo repetidamente la lista a la mitad hasta encontrar el elemento deseado o determinar que no está presente.

Este es otro que usa objetivo o target y es el algoritmo de búsqueda mas largo que hemos visto, este usa while y se parece a Merge sort pero con la diferencia de que este ocupa un target para buscar

**Poder identificar un algoritmo por su Código**

**- Inserción**

public class InsertionSortExample {

    public static int insertionSort(int arr[]) { //aqui se crea la clase de insertion sort para luego poder llamarla

        int n = arr.length;

        int operaciones = 0; //el contador sirve para ver el numero de comparaciones que se hace

        for (int i = 1; i < n; i++) {// este bucle sirve para recorrer el arreglo y compararlo para saber cual mover

            int key = arr[i];//se usa la variable key para guardar los valores y poder compararlos

            int j = i - 1;

            while (j >= 0 && arr[j] > key) {// este while sirve para recorrer los valores mas grandes hacia la derecha

                arr[j + 1] = arr[j];

                j = j - 1;

                operaciones++;//el contador ve los movimientos y los va guardando

            }

            arr[j + 1] = key;

        }

        return operaciones;

    }

}

**- Selección**

public class SelectionSortExample {

    public static int selectionSort(int arr[]) {

        int n = arr.length;

        int comparisons = 0;

        for (int i = 0; i < n - 1; i++) { //aqui se inicia el bucle for n representa el valor que sera cambiado de pocision, es -1 porque ahi quedara el dato ordenadp

            int minIndex = i;//miniindex guarda el elemento mas pequeno

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

                comparisons++;

                if (arr[j] < arr[minIndex]) {

                    minIndex = j;//cuando se inicia el bucle se comparan todos los valores del arreglo y agarra el mas pequeno y lo mete en miniindex osea j para que pueda ser cambiado

                }

            }

            // Intercambio de elementos

            int temp = arr[minIndex];

            arr[minIndex] = arr[i];

            arr[i] = temp;//Si minIndex cambió, significa que se encontró un número menor y hay que intercambiarlo con arr[i].

        }

        return comparisons;

    }

}

**- Bubble Sort**

public class BubbleSortExample {

    public static int bubbleSort(int[] arr) { //se crea la clase bubblesort

        int n = arr.length; //aqui se usa int para que el valor sea un entero y se usa lenght para ver el tamano del arreglo

        int comparisons = 0; // este es un contador para poder ver el numero de comparaciones del programa

        boolean swapped;

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

            swapped = false;

            for (int j = 0; j < n - 1 - i; j++) { //aqui se usa un ciclo for para el bubble sort en el que inicia en 0 en el que se comparan datos y se intercambian elementos dentro de un arreglo

                comparisons++; // aqui se cuenta el numero de comparaciones

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

                    int temp = arr[j];

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

                    arr[j + 1] = temp;

                    swapped = true;

                }

            }

            if (!swapped) break; // Si no hubo intercambios, ya está ordenado

        }

        return comparisons;

    }

}

**- Merge Sort**

public class MergeSort {

    public int operationCount = 0; // Contador de operaciones

    public int comparisonCount = 0; // Contador de comparaciones

    public void mergeSort(int[] arr, int left, int right) {

        if (left < right) {

            int mid = left + (right - left) / 2;

            mergeSort(arr, left, mid);

            mergeSort(arr, mid + 1, right);

            merge(arr, left, mid, right);

        }

    }

    public void merge(int[] arr, int left, int mid, int right) {

        int n1 = mid - left + 1;

        int n2 = right - mid;

        int[] leftArray = new int[n1];

        int[] rightArray = new int[n2];

        // Copiamos los datos en arreglos temporales

        for (int i = 0; i < n1; i++) {

            leftArray[i] = arr[left + i];

            operationCount++; // Contar copia

        }

        for (int j = 0; j < n2; j++) {

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

            operationCount++; // Contar copia

        }

        // Fusionamos las dos mitades ordenadas

        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {

            comparisonCount++; // Contar comparación

            if (leftArray[i] <= rightArray[j]) {

                arr[k] = leftArray[i];

                i++;

            } else {

                arr[k] = rightArray[j];

                j++;

            }

            k++;

            operationCount++; // Contar asignación

        }

        // Copiar los elementos restantes de leftArray[], si hay alguno

        while (i < n1) {

            arr[k] = leftArray[i];

            i++;

            k++;

            operationCount++; // Contar asignación

        }

        // Copiar los elementos restantes de rightArray[], si hay alguno

        while (j < n2) {

            arr[k] = rightArray[j];

            j++;

            k++;

            operationCount++; // Contar asignación

        }

    }

    public void printResults() {

        System.out.println("Total de operaciones: " + operationCount);

        System.out.println("Total de comparaciones: " + comparisonCount);

    }

}

**- Quick Sort**

import java.util.Arrays;

public class QuickSort {

    static int comparaciones = 0;

    static int intercambios = 0;

    public void swap(int[] arr, int i, int j) {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

        intercambios++;

    }

    // Partición usando el esquema de partición de Lomuto

    public int partition(int[] a, int start, int end) {

        int pivot = a[end];

        int pIndex = start;

        for (int i = start; i < end; i++) {

            comparaciones++;

            if (a[i] <= pivot) {

                swap(a, i, pIndex);

                pIndex++;

            }

        }

        swap(a, end, pIndex);

        return pIndex;

    }

    public void quicksort(int[] a, int start, int end) {

        if (start >= end) {

            return;

        }

        int pivot = partition(a, start, end);

        quicksort(a, start, pivot - 1);

        quicksort(a, pivot + 1, end);

    }

}

**- búsqueda secuencial**

public class BusquedaSecuencial {

    public static int buscar(int[] arreglo, int objetivo) {

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

            if (arreglo[i] == objetivo) {

                return i; // Retorna la posición del elemento encontrado

            }

        }

        return -1; // Retorna -1 si el elemento no está en el arreglo

    }

    }

**- búsqueda Binaria**

public class BinarySearch {

    public static int search (int[] arrayb, int  target) {

        int left = 0, right = arrayb.length - 1;

        while (left <= right) {

            int middle = left + (right - left) / 2;

            if (arrayb[middle] == target) {

                return middle; // Elemento encontrado

            }

            if (arrayb[middle] < target) {

                left = middle + 1;

            } else {

                right = middle - 1;

            }

        }

        return -1; //si no se encuentra el valor, se regresa -1

    }

}

**Poder calcular el peor caso para cada uno de los algoritmos con la pura formula**

**Captura de pantalla con letras

El contenido generado por IA puede ser incorrecto.**

**Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.**

log2 en todos los casos

**- Inserción**

(5) = 5 \* (5-1)/2 = 10

**- Selección**

(10) = 10 \* (10-1)/2 = 45

**- Bubble Sort**

(15) = 15 \* (15-1)/2 = 105

**- Merge Sort**

(20) = 20 \* log2 20 = 87

**- Quick Sort**

(25) = 25 \* (25-1)/2 = 300

**- búsqueda secuencial**

(30) = 30

**- búsqueda Binaria**

(35) = log2 35 = 6

**Poder realizar una prueba de escritorio para cada uno de los siguientes algoritmos con datos de entrada pequeño (5 elementos)**

**- Inserción**

**- Selección**

**- Bubble Sort**

**- Merge Sort**

**- Quick Sort**

**- búsqueda secuencial**

**- búsqueda Binaria**