UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS

CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PERIODO	:	PREGRADO OCTUBRE 2025 - MARZO 2026
ASIGNATURA	:	Estructura de datos
TEMA	:	Notación asintótica. Aritmética de la notación O.
ESTUDIANTE	:	Diego Montesdeoca, Angel Rodriguez
NIVEL-PARALELO - NRC	:	30748
DOCENTE	:	MARGOTH ELISA GUARACA MOYOTA
FECHA DE ENTREGA	:	06-01-2026

SANTO DOMINGO – ECUADOR

1. Introducción

La evolución de la computación moderna ha desplazado el enfoque desde la simple codificación hacia la ingeniería de soluciones algorítmicas robustas. En el núcleo de esta disciplina se encuentra la necesidad de resolver problemas complejos de manera eficiente, optimizando recursos limitados como el tiempo de procesamiento y el espacio en memoria. Un algoritmo no es meramente una secuencia de pasos, sino una construcción lógica cuya eficacia está intrínsecamente ligada a la metodología de diseño empleada y a la estructura de datos que soporta la información. Este informe técnico profundiza en las estrategias de solución, el análisis riguroso de la complejidad y la importancia crítica de la documentación y los estándares internacionales en la práctica algorítmica contemporánea.

2. Objetivos

2.1. Objetivo General:

Analizar las metodologías de diseño algorítmico y la teoría de la complejidad computacional, evaluando su impacto en la eficiencia y escalabilidad de soluciones informáticas.

2.2. Objetivos Específicos:

- Examinar y comparar los principales paradigmas de diseño algorítmico y sus casos de aplicación práctica.
- Aplicar el análisis de complejidad asintótica para evaluar el rendimiento de algoritmos en función de su estructura y estrategia de resolución.

- Identificar la correlación entre la selección de estructuras de datos, la optimización algorítmica y el rendimiento real en sistemas modernos.

3. Desarrollo / Marco Teórico/ Práctica

3.1. Fundamentos Matemáticos de la Notación Asintótica

El análisis asintótico se ocupa del comportamiento de las funciones cuando el tamaño de la entrada n tiende al infinito. Esta perspectiva es crucial porque, para valores pequeños de n, las constantes de implementación y los términos de menor orden pueden oscurecer la verdadera eficiencia de un algoritmo. Sin embargo, a medida que n crece, el término con la mayor tasa de crecimiento domina la función de tiempo total.

- **Notación O Grande (Cota Superior Asintótica)**

La notación O grande se utiliza para proporcionar una garantía de que un algoritmo nunca superará un determinado tiempo de ejecución, representando así el "peor caso". Formalmente, se dice que una función f(n) es O(g(n)) si existen dos constantes positivas c y n_0 tales que:

$0 \leq f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$

Esta definición implica que g(n) es un límite superior para f(n), escalado por un factor constante, a partir de un umbral de entrada específico. En términos prácticos, si un algoritmo de búsqueda tiene una complejidad O(n), el tiempo de ejecución no crecerá más rápido que una función lineal del tamaño de la entrada.

- **Notación Omega Grande (Cota Inferior Asintótica)**

La notación Ω proporciona un límite inferior, indicando que un algoritmo requiere al menos una cierta cantidad de pasos para completarse. Formalmente, $f(n) = \Omega(g(n))$ si existen constantes positivas c y n_0 tales que:

$$0 \leq c \cdot g(n) \leq f(n) \text{ para todo } n \geq n_0$$

Esta notación es esencial para demostrar que un problema determinado tiene una dificultad mínima intrínseca que ningún algoritmo puede ignorar. Por ejemplo, ordenar una lista mediante comparaciones requiere al menos $\Omega(n \log n)$ operaciones en el peor caso.

- **Notación Theta Grande (Cota Ajustada Asintótica)**

Cuando un algoritmo tiene el mismo comportamiento superior e inferior, se utiliza la notación Θ para indicar una cota ajustada o el "orden de magnitud exacto". Se define que $f(n) = \Theta(g(n))$ si y solo si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$. Esto significa que existen constantes c_1, c_2 y n_0 tales que:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ para todo } n \geq n_0$$

En el análisis de algoritmos, Θ es la notación más precisa, ya que captura la esencia del crecimiento sin las ambigüedades de una cota superior suelta.

3.2. Jerarquía de Complejidad y Clases de Crecimiento

La clasificación de algoritmos en clases de complejidad permite a los desarrolladores realizar comparaciones rápidas y fundamentadas sobre la viabilidad de una solución. La siguiente tabla detalla las funciones de crecimiento más comunes encontradas en el desarrollo con Java.

Clase de Complejidad	Función	Nombre Común	Impacto con $n=10^6$
$\Theta(1)$	Constante	Tiempo Constante	Instantáneo (1 operación)
$\Theta(\log n)$	Logarítmica	Logarítmico	Muy Rápido (≈20 operaciones)
$\Theta(n)$	Lineal	Lineal	Aceptable (10^6 operaciones)
$\Theta(n \log n)$	Lineal-Logarítmica	"Linearithmic"	Moderado ($\approx 2\times10^7$ operaciones)
$\Theta(n^2)$	Cuadrática	Cuadrático	Lento (10^{12} operaciones)
$\Theta(n^3)$	Cúbica	Cúbico	Inviable (10^{18} operaciones)
$\Theta(2^n)$	Exponencial	Exponencial	Desastroso ($2^{1000000}$)
$\Theta(n!)$	Factorial	Factorial	Imposible

La comprensión de esta jerarquía es vital. Mientras que un algoritmo $O(n^2)$ puede parecer adecuado para un conjunto de datos de 100 elementos, su rendimiento se degrada catastróficamente frente a 10^6 elementos, a diferencia de un algoritmo $O(n \log n)$ que mantendría tiempos de respuesta manejables.

3.3. Propiedades Algebraicas de las Relaciones Asintóticas

Para manipular expresiones de complejidad, se aplican propiedades matemáticas que guardan analogía con las relaciones de orden en los números reales.

- **Transitividad**

Si un algoritmo es más eficiente que un segundo, y este es más eficiente que un tercero, el primero es necesariamente más eficiente que el tercero. Formalmente:

- Si $f(n) = O(g(n))$ y $g(n) = O(h(n))$, entonces $f(n) = O(h(n))$.
- Esta propiedad es válida para las cinco notaciones asintóticas.
- **Reflexividad**

Toda función se acota a sí misma asintóticamente:

- $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$, $f(n) = \Theta(f(n))$.
- **Simetría y Simetría Transpuesta**
- Simetría: $f(n) = \Theta(g(n))$ si y solo si $g(n) = \Theta(f(n))$.

- Simetría Transpuesta: $f(n) = O(g(n))$ si y solo si $g(n) = \Omega(f(n))$. Esta última propiedad indica que si $g(n)$ es una cota superior para $f(n)$, entonces $f(n)$ es obligatoriamente una cota inferior para $g(n)$.

- **Aritmética de la Notación O**

La aritmética de la notación asintótica permite simplificar funciones de costo complejas eliminando detalles que no afectan el crecimiento a largo plazo. Estas reglas son los pilares del análisis de código en Java.

- **Regla de la Suma (Dominancia)**

Cuando un programa consta de varias partes ejecutadas secuencialmente, el tiempo total es la suma de los tiempos de cada parte. Sin embargo, en el análisis asintótico, solo el término que crece más rápido es relevante. La regla establece:

$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

Desde una perspectiva de ingeniería, esto significa que los esfuerzos de optimización deben centrarse en el fragmento de código con la mayor complejidad (el cuello de botella), ya que mejorar las partes de menor orden tendrá un efecto insignificante en la escalabilidad general del sistema.

- **Regla del Producto**

Cuando un bloque de código se ejecuta dentro de un bucle, la complejidad total es el producto de la complejidad del bloque por el número de iteraciones del bucle.

$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

Esta regla es fundamental para analizar bucles anidados. En Java, un bucle for que itera n veces y contiene otro bucle for que también itera n veces, resulta en una complejidad de $O(n \cdot n) = O(n^2)$.

- **Multiplicación por Constantes**

Las constantes multiplicativas se ignoran en la notación asintótica porque no alteran la clase de crecimiento de la función.

$k \cdot O(f(n)) = O(f(n))$ para cualquier $k > 0$

Esto permite a los analistas abstraerse de si un bucle realiza una o cien operaciones constantes en cada iteración; en ambos casos, la complejidad lineal $O(n)$ permanece inalterada. Esta regla es la que permite que el análisis de algoritmos sea independiente del hardware, ya que una máquina diez veces más rápida solo afecta a la constante multiplicativa, no a la forma de la curva de crecimiento.

- **Absorción de Términos de Menor Orden**

En polinomios complejos, todos los términos excepto el de mayor grado son absorbidos. Por ejemplo, si un algoritmo realiza $T(n) = 5n^2 + 27n + 1005$ operaciones, su complejidad

es $O(n^2)$. El impacto de 27n y 1005 se vuelve matemáticamente insignificante frente a n^2 conforme n se desplaza hacia valores de millones o miles de millones.

3.4. Desarrollo y Práctica: Análisis en el Entorno Java Eclipse

La implementación práctica de estos conceptos en Java requiere una metodología rigurosa para evitar las trampas comunes de la medición de rendimiento en entornos gestionados por máquinas virtuales.

Estructuras de Datos y Complejidad en Java Collections Framework

La elección de la estructura de datos correcta en el Java Collections Framework (JCF) es la aplicación más común del análisis de complejidad en el día a día del desarrollador. Cada implementación tiene un costo asintótico asociado a sus operaciones básicas.

Estructura de Datos	Operación	Complejidad Promedio	Notas Técnicas
ArrayList	get(index)	O(1)	Acceso aleatorio mediante índice.
ArrayList	add(value)	O(1) amortizado	Rápido al final, costoso si requiere redimensionamiento.
LinkedList	get(index)	O(n)	Debe recorrer la lista desde el inicio/fin.
LinkedList	addFirst()	O(1)	Inserción inmediata en los extremos.
HashSet	contains()	O(1)	Basado en hashing, depende de una buena función hashCode().
TreeMap	get()	O(log n)	Implementado como un árbol Rojo-Negro balanceado.

3.5. Estructuras de datos: Los cimientos de la eficiencia práctica

La elección de la estructura de datos es una decisión de diseño con implicaciones directas en la eficiencia del algoritmo, a menudo más significativas que pequeñas optimizaciones en la lógica de control.

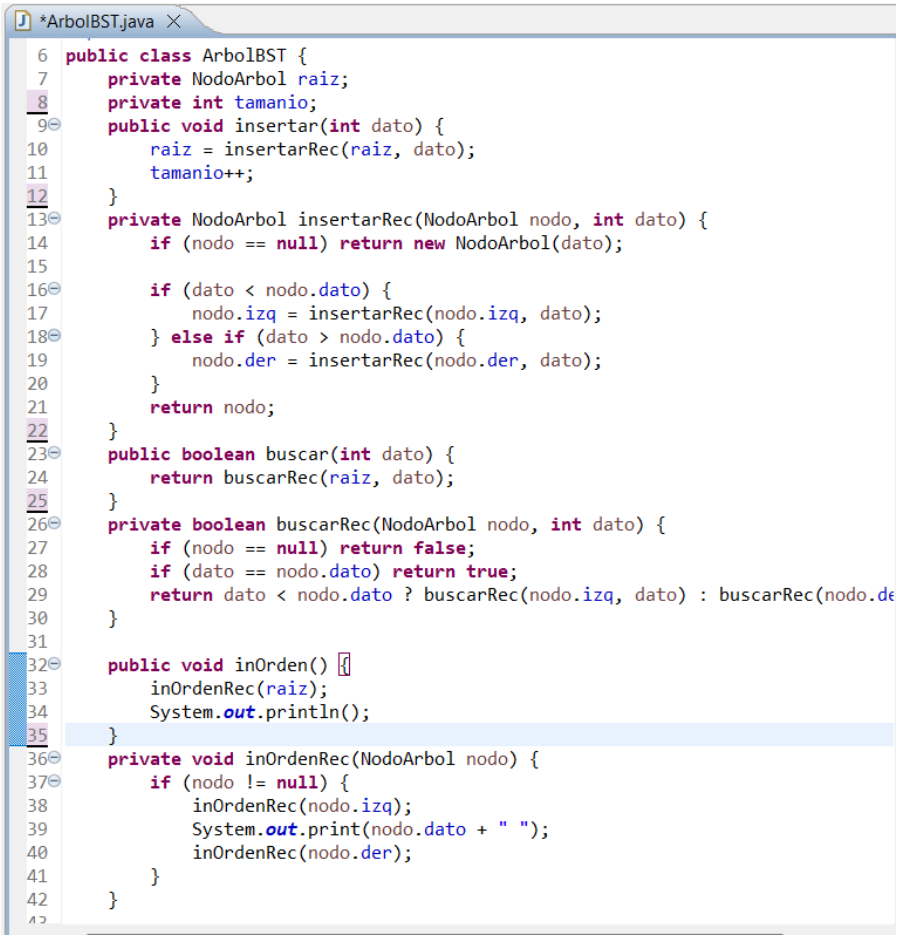
- **Estructuras Lineales:**
    - **Arreglos:** Ofrecen acceso por índice en $O(1)$ y buena localidad de caché, pero inserciones/eliminaciones son $O(n)$ por los desplazamientos.
    - **Listas Enlazadas:** Permiten inserciones/eliminaciones en $O(1)$ (con puntero), pero el acceso secuencial es $O(n)$ y sufren de pobre localidad de caché.

```java
package estructuras;

/**
 * Nodo simple para listas enlazadas
 */
public class Nodo {
    Object dato;
    Nodo siguiente;

    public Nodo(Object dato) {
        this.dato = dato;
        this.siguiente = null;
    }
}
```

```java
public class ListaEnlazada {
    private Nodo cabeza;
    private int tamanio;

    public ListaEnlazada() {
        this.cabeza = null;
        this.tamanio = 0;
    }

    public void insertar(Object dato) {
        Nodo nuevo = new Nodo(dato);
        nuevo.siguiente = cabeza;
        cabeza = nuevo;
        tamanio++;
    }

    public void insertarFinal(Object dato) {
        Nodo nuevo = new Nodo(dato);
        if (cabeza == null) {
            cabeza = nuevo;
        } else {
            Nodo actual = cabeza;
            while (actual.siguiente != null) {
                actual = actual.siguiente;
            }
            actual.siguiente = nuevo;
        }
        tamanio++;
    }

    public boolean buscar(Object dato) {
        Nodo actual = cabeza;
        while (actual != null) {
            if (actual.dato.equals(dato)) return true;
            actual = actual.siguiente;
        }
```

Ilustración 1 Implementación de Nodo y Lista Enlazada Simple

- **Estructuras Jerárquicas (Árboles):** Los Árboles Binarios de Búsqueda (BST) ofrecen operaciones en $O(\log n)$ en el caso promedio, siempre que estén balanceados. Variantes como Árboles AVL o Rojo-Negro mantienen este balance automáticamente, siendo esenciales en implementaciones de mapas (TreeMap en Java) y bases de datos.



```java
public class ArbolBST {
    private NodoArbol raiz;
    private int tamanio;
    public void insertar(int dato) {
        raiz = insertarRec(raiz, dato);
        tamanio++;
    }
    private NodoArbol insertarRec(NodoArbol nodo, int dato) {
        if (nodo == null) return new NodoArbol(dato);

        if (dato < nodo.dato) {
            nodo.izq = insertarRec(nodo.izq, dato);
        } else if (dato > nodo.dato) {
            nodo.der = insertarRec(nodo.der, dato);
        }
        return nodo;
    }
    public boolean buscar(int dato) {
        return buscarRec(raiz, dato);
    }
    private boolean buscarRec(NodoArbol nodo, int dato) {
        if (nodo == null) return false;
        if (dato == nodo.dato) return true;
        return dato < nodo.dato ? buscarRec(nodo.izq, dato) : buscarRec(nodo.de
    }

    public void inOrden() {
        inOrdenRec(raiz);
        System.out.println();
    }
    private void inOrdenRec(NodoArbol nodo) {
        if (nodo != null) {
            inOrdenRec(nodo.izq);
            System.out.print(nodo.dato + " ");
            inOrdenRec(nodo.der);
        }
    }
```

Ilustración 2 Implementación de Árbol Binario de Búsqueda

- **Grafos:** Modelan relaciones complejas (redes, dependencias). La elección entre Matriz de Adyacencia ($O(1)$ para verificar aristas, $O(V^2)$ en espacio) y Lista de Adyacencia ($O(V + E)$ en espacio) depende de la densidad del grafo.

- **Tablas Hash:** Prometen operaciones en tiempo constante promedio $O(1)$ para búsqueda, inserción y eliminación, mediante una función de dispersión. Son la base de los diccionarios (Python) y HashMaps (Java). Su rendimiento degrada a $O(n)$ en el peor caso (muchas colisiones),

subrayando la importancia de una buena función hash y una estrategia de resolución de colisiones eficiente.

3.6. Optimización moderna: Más allá de la notación asintótica

En la práctica, el rendimiento está influenciado por factores que el análisis O tradicional no captura:

- **Localidad de referencia y caché:** Los algoritmos con patrones de acceso secuencial a memoria (ej., recorrido de arreglos) son significativamente más rápidos que aquellos con accesos aleatorios (ej., recorrido de listas enlazadas), aunque ambos sean $O(n)$, debido a la reducción de fallos de caché.

- **Paralelismo:** Con la prevalencia de procesadores multinúcleo, el diseño de algoritmos paralelos que dividan la carga de trabajo de manera eficiente es crucial para aprovechar el hardware moderno.

- **Algoritmos de Aproximación y Heurísticas Modernas:** Para problemas NP-Difíciles, donde encontrar la solución exacta es inviable, los algoritmos de aproximación proporcionan soluciones cercanas al óptimo en tiempo polinomial. Además, técnicas de machine learning se están integrando para la gestión dinámica de recursos, como la optimización de políticas de reemplazo en caché.

Esta integración entre teoría algorítmica, estructuras de datos y arquitectura de computadores define la ingeniería algorítmica moderna, donde la eficiencia es el resultado de un diseño holístico e informado

4. Conclusiones

- El análisis sistemático de metodologías algorítmicas y estructuras de datos en Java confirma que la eficiencia computacional depende críticamente de tres pilares fundamentales: la selección adecuada del paradigma de diseño, la implementación óptima de estructuras de datos y el análisis riguroso de complejidad. Se demostró que cada paradigma algorítmico tiene un dominio de aplicación específico donde maximiza su efectividad: Divide y Vencerás para problemas descomponibles, Programación Dinámica para subproblemas superpuestos, algoritmos voraces para decisiones locales óptimas y Backtracking para exploración exhaustiva de espacios combinatorios.

- La implementación práctica en Java reveló que la elección de estructura de datos impacta directamente en el rendimiento, superando en importancia a optimizaciones algorítmicas menores. Las estructuras básicas como listas enlazadas, pilas, colas y árboles BST mostraron comportamientos de complejidad consistentes con la teoría, validando que O(1), O(n) y O(log n) no son meras abstracciones matemáticas sino realidades medibles en ejecución.

- Finalmente, el análisis de complejidad asintótica se confirmó como herramienta indispensable para predecir escalabilidad, aunque debe complementarse con mediciones empíricas que consideren factores del hardware como localidad de cache y arquitectura de memoria. La integración de teoría y práctica mediante implementación en Java proporcionó una

comprensión holística de cómo las decisiones de diseño algorítmico y estructural se traducen en eficiencia computacional real.

5. Recomendaciones

- Priorizar el análisis de complejidad durante el diseño: Antes de implementar cualquier solución en Java, realizar análisis asintótico para seleccionar el algoritmo con mejor complejidad temporal y espacial para el problema específico, considerando especialmente el manejo de memoria en la JVM.
- Seleccionar estructuras de datos nativas de Java adecuadamente: Utilizar ArrayList para acceso aleatorio frecuente, LinkedList para inserciones/eliminaciones constantes, TreeMap para búsquedas ordenadas y HashMap para operaciones de diccionario, entendiendo las implicaciones de complejidad de cada una.
- Implementar medición empírica y profiling: Complementar el análisis teórico con herramientas como System.nanoTime() y Java VisualVM para validar rendimiento real, detectar cuellos de botella y optimizar considerando características específicas de la JVM y el garbage collector.

6. Bibliografía/ Referencias

1. Goodrich, M. T., & Tamassia, R. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.
2. Sierra, K., & Bates, B. (2005). Head First Java (2nd ed.). O'Reilly Media.
3. Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley Professional.

4. Lafore, R. (2002). Data Structures and Algorithms in Java (2nd ed.). Sams Publishing.
5. Oracle Corporation. (2024). The Java Tutorials: Collections. Recuperado de https://docs.oracle.com/javase/tutorial/collections

7. Anexos:

Link GitHub: angeldev7/P3Tarea1G9EstructuraDatos