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	:	Metodologías de solución algorítmica y análisis de Algoritmos
ESTUDIANTE	:	Diego Montesdeoca, Angel Rodriguez
NIVEL-PARALELO - NRC	:	30748
DOCENTE	:	MARGOTH ELISA GUARACA MOYOTA
FECHA DE ENTREGA	:	06-01-2026

SANTO DOMINGO – ECUADOR

Tabla de contenido

Tabla de ilustraciones

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. Metodologías de resolución de problemas: De Pólya a la ingeniería de software

La resolución sistemática de problemas en ciencias de la computación trasciende la intuición programática, estructurándose en metodologías formalizadas que garantizan corrección, eficiencia y mantenibilidad. El método heurístico de George Pólya, aunque originado en matemáticas, constituye la base cognitiva del pensamiento algorítmico moderno. Sus cuatro fases —comprensión del problema, concepción de un plan, ejecución y visión retrospectiva— proporcionan un ciclo iterativo que guía desde la abstracción del enunciado hasta la validación y optimización de la solución.

En el contexto de la ingeniería de software, esta heurística se materializa en dos enfoques estructurales fundamentales:

- **Diseño Top-Down (Descendente):** Parte de una visión de alto nivel, descomponiendo funcionalidades complejas en módulos más simples y manejables. Este enfoque promueve la modularidad, facilita la asignación de tareas en equipos de desarrollo y permite la creación de "stubs" para pruebas tempranas.

- **Diseño Bottom-Up (Ascendente):** Comienza con la implementación de componentes básicos y reutilizables (como librerías de funciones o clases fundamentales), que posteriormente se integran para construir sistemas más

complejos. Es especialmente útil en contextos exploratorios o cuando se prioriza la creación de utilidades genéricas.

La elección entre uno u otro enfoque no es excluyente; en la práctica, se suelen combinar, utilizando *Top-Down* para el diseño arquitectónico y *Bottom-Up* para la implementación de componentes críticos o ya existentes.

3.2. Paradigmas algorítmicos: Estrategias fundamentales para la eficiencia

La eficiencia de un algoritmo está profundamente ligada al paradigma de diseño seleccionado. Cada paradigma representa una filosofía distinta para abordar la descomposición y resolución de problemas:

- **Divide y Vencerás (Divide and Conquer):** Se fundamenta en la recursividad y la descomposición. El problema se divide en subproblemas independientes de menor tamaño, estos se resuelven (generalmente de la misma manera), y sus soluciones se combinan para formar la solución global. Su expresión matemática se modela mediante relaciones de recurrencia. Ejemplos paradigmáticos incluyen Merge Sort ($T(n) = 2T(n/2) + n, \Theta(n\log\, n)$) y el algoritmo de Strassen para multiplicación de matrices, que demuestra cómo una división inteligente puede superar límites teóricos aparentes (reduciendo $O(n^3)$ a ~$O(n^{2.807})$).

```
18    public static void mergeSort(int[] arreglo) {
19        if (arreglo == null || arreglo.length <= 1) {
20            return;
21        }
22        mergeSortRecursivo(arreglo, 0, arreglo.length - 1);
23    }
24
```

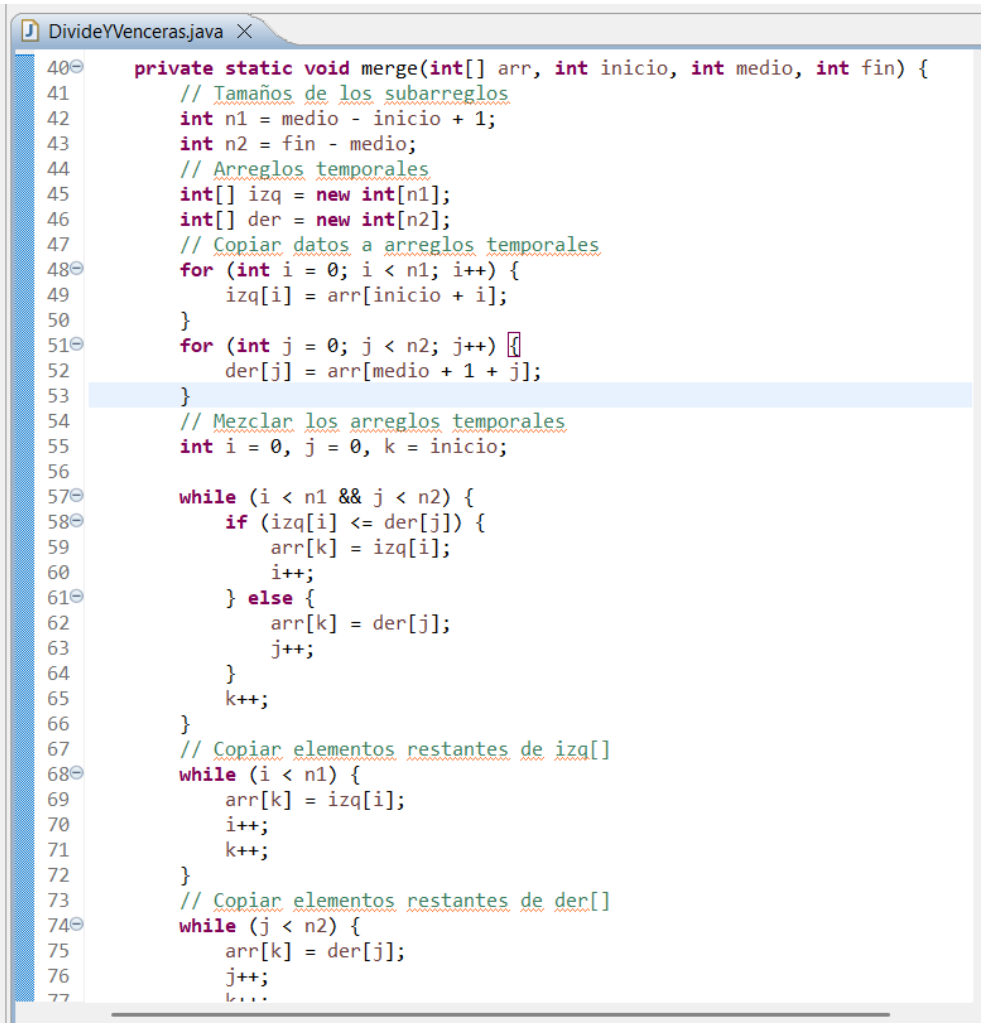


Ilustración 1Algoritmo Merge Sort: Implementación Divide y Vencerás

- **Programación Dinámica:** Surge como respuesta a la ineficiencia de la recursión simple ante subproblemas superpuestos. En lugar de recalcular soluciones, estas se almacenan en una tabla (tabulación) o mediante memorización (memoization). Este paradigma es óptimo para problemas de optimización con subestructura óptima, como el problema de la mochila 0-1 o el cálculo eficiente de la secuencia de Fibonacci. Garantiza la solución óptima a cambio de un mayor consumo de espacio, ilustrando la clásica disyuntiva tiempo vs. espacio.

```java
    public static long fibonacciMemoization(int n) {
        long[] memo = new long[n + 1];
        for (int i = 0; i <= n; i++) {
            memo[i] = -1;
        }
        return fibMemo(n, memo);
    }
```

```java
    public static long fibonacciTabulacion(int n) {
        if (n <= 1) {
            return n;
        }

        long[] tabla = new long[n + 1];
        tabla[0] = 0;
        tabla[1] = 1;

        for (int i = 2; i <= n; i++) {
            tabla[i] = tabla[i - 1] + tabla[i - 2];
        }

        return tabla[n];
    }
```

Ilustración 2 Fibonacci con Programación Dinámica: Memoization vs Tabulación

- **Algoritmos Voraces (Greedy):** Toman decisiones locales óptimas en cada paso, sin reconsideraciones posteriores, con la esperanza de alcanzar un óptimo global.

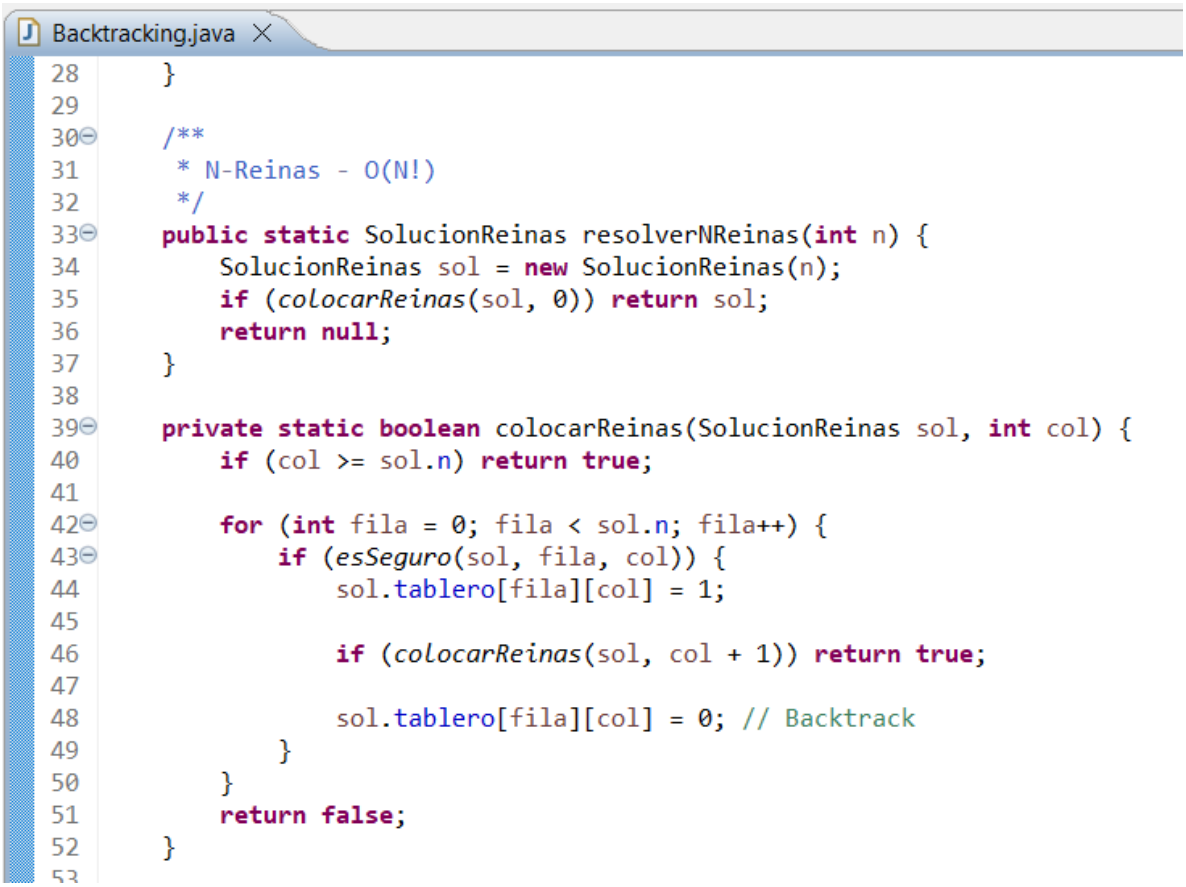
AlgoritmosVoraces.java

```java
    */
    public static String cambioVoraz(int[] denoms, int cantidad) {
        ordenarDesc(denoms);
        int totalMonedas = 0;
        String resultado = "";

        for (int d : denoms) {
            if (cantidad >= d) {
                int num = cantidad / d;
                totalMonedas += num;
                resultado += num + " x $" + d + "\n";
                cantidad %= d;
            }
        }

        if (cantidad > 0) return null;
        return "Total: " + totalMonedas + " monedas\n" + resultado;
    }
```

Ilustración 3 Algoritmo Voraz: Cambio de Monedas

Su correctitud requiere que el problema cumpla con la propiedad de la elección voraz y una subestructura óptima. Son notablemente eficientes pero no siempre garantizan la solución óptima absoluta (ejemplo clásico: el problema del cambio con sistemas monetarios arbitrarios). Algoritmos como Dijkstra (caminos más cortos) y Kruskal (árbol de expansión mínima) son pilares de este paradigma.

Backtracking (Vuelta Atrás): Es una técnica de fuerza bruta inteligente que explora de forma sistemática el espacio de soluciones, modelado como un árbol. Realiza un recorrido en profundidad (DFS), construyendo soluciones parciales. Si una rama no conduce a una solución válida, el algoritmo retrocede al nodo anterior para probar alternativas.



```java
    }

    /**
     * N-Reinas - O(N!)
     */
    public static SolucionReinas resolverNReinas(int n) {
        SolucionReinas sol = new SolucionReinas(n);
        if (colocarReinas(sol, 0)) return sol;
        return null;
    }

    private static boolean colocarReinas(SolucionReinas sol, int col) {
        if (col >= sol.n) return true;

        for (int fila = 0; fila < sol.n; fila++) {
            if (esSeguro(sol, fila, col)) {
                sol.tablero[fila][col] = 1;

                if (colocarReinas(sol, col + 1)) return true;

                sol.tablero[fila][col] = 0; // Backtrack
            }
        }
        return false;
    }
```

Ilustración 4 Backtracking: Solución de N-Reinas

Es fundamental para problemas combinatorios, de satisfacción de restricciones y juegos (como el de las N-Reinas o Sudoku), donde no existe una fórmula directa.

3.3. Complejidad computacional: El lenguaje de la escalabilidad

El análisis de algoritmos trasciende la mera medición empírica, estableciendo un marco teórico para predecir el comportamiento asintótico conforme el tamaño de la entrada (n) tiende a infinito. Se evalúan dos dimensiones:

- Complejidad Temporal: Número de operaciones elementales ejecutadas.
- Complejidad Espacial: Cantidad de memoria total utilizada.
- Las notaciones asintóticas proporcionan el lenguaje para este análisis:

- $O(g(n))$ (Big O - Cota Superior): Define el límite superior del crecimiento en el peor caso. Garantiza que el algoritmo no superará ese ritmo de consumo de recursos.
- $\Omega(g(n))$ (Omega - Cota Inferior): Define el límite inferior en el mejor caso, indicando el mínimo de recursos necesarios.
- $\Theta(g(n))$ (Theta - Cota Ajustada): Se emplea cuando las cotas superior e inferior coinciden, describiendo con precisión el crecimiento exacto del algoritmo.

```java
    /**
     * Mide el tiempo de ejecución de una operación
     */
    public static long medirTiempo(Runnable operacion) {
        long inicio = System.nanoTime();
        operacion.run();
        long fin = System.nanoTime();
        return fin - inicio;
    }
```

AnalizadorComplejidad.java

```java
/**
 * Compara el rendimiento de dos algoritmos
 */
public static void compararAlgoritmos(String nombre1, Runnable alg1,
                                      String nombre2, Runnable alg2) {
    System.out.println("\n╔═══════════════════════════════════════════════════════════╗");
    System.out.println("║        COMPARACIÓN DE RENDIMIENTO DE ALGORITMOS           ║");
    System.out.println("╚═══════════════════════════════════════════════════════════╝");

    System.out.println("\nEjecutando " + nombre1 + "...");
    long tiempo1 = medirTiempo(alg1);

    System.out.println("Ejecutando " + nombre2 + "...");
    long tiempo2 = medirTiempo(alg2);

    System.out.println("\n┌─────────────────────────────────────────────────────────┐");
    System.out.println("│ RESULTADOS:                                             │");
    System.out.println("├─────────────────────────────────────────────────────────┤");
    System.out.printf("│ %-30s : %20s │\n", nombre1, formatearTiempo(tiempo1));
    System.out.printf("│ %-30s : %20s │\n", nombre2, formatearTiempo(tiempo2));
    System.out.println("├─────────────────────────────────────────────────────────┤");

    if (tiempo1 < tiempo2) {
        double factor = (double) tiempo2 / tiempo1;
        System.out.printf("│ %s es %.2fx más rápido                  │\n",
            nombre1, factor);
    } else {
        double factor = (double) tiempo1 / tiempo2;
        System.out.printf("│ %s es %.2fx más rápido                  │\n",
            nombre2, factor);
    }
    System.out.println("└─────────────────────────────────────────────────────────┘");
}
```

Ilustración 5 Clase Analizador de Complejidad Computacional

La jerarquía de complejidades comunes ($O(1)$, $O(\log\ n)$, $O(n)$, $O(n\log\ n)$, $O(n^2)$, $O(2^n)$, $O(n!)$) sirve como mapa para clasificar algoritmos y descartar soluciones inviables para entradas grandes. Un principio clave es el del término dominante: en una función como $f(n) = 5n^2 + 100n + 300$, la complejidad es $\Theta(n^2)$, pues el término cuadrático domina el crecimiento para valores grandes de n.

3.4. 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.

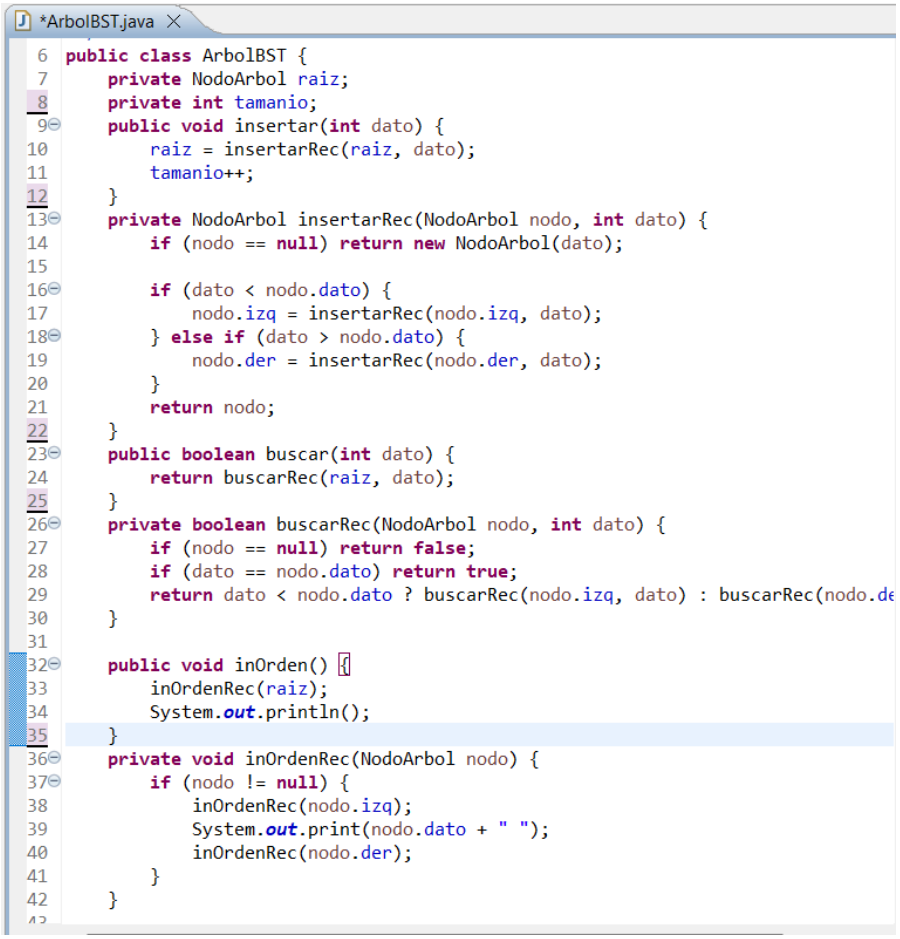
- **Estructuras Lineales:**
    - **Arreglos:** Ofrecen acceso por índice en $O(1)$ y buena localidad de caché, pero inserciones/eliminaciones son $O(n)$ por los desplazamientos.
    - **Listas Enlazadas:** Permiten inserciones/eliminaciones en $O(1)$ (con puntero), pero el acceso secuencial es $O(n)$ 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 6 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 7 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.

3.5. 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