Resumen - Estructura de Datos y Algoritmos

Lucas Solar Grillo

February 17, 2025

Contents

1 Primer parcial

1.1 Introducción

1.1.1 Conceptos Clave

- **Algoritmo**: Conjunto de pasos definidos para resolver un problema computacional.
- **Estructura de datos**: Método de organización y almacenamiento de datos.
- **Notación Big O**: Clasifica algoritmos según su uso de recursos.

1.1.2 Ejemplo Numérico de Complejidad

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	1	0	1	0	1
10	1	3	10	33	100
100	1	7	100	664	10000

1.1.3 Ejemplo: Algoritmo Fibonacci

Recursivo (Ineficiente, $O(2^n)$)

```
int fibonacci(int n) {
    if (n <= 0) return 0;
    if (n <= 2) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Iterativo (Eficiente, $O(n)$)

```
int fibonacci(int n) {
    int prev = 0, curr = 1;
    for (int i = 1; i < n; i++) {
        int next = prev + curr;
        prev = curr;
        curr = next;
    }
    return (n <= 0) ? prev : curr;
}
```

1.2 C++ y STL

1.2.1 Conceptos Clave

- **Abstracción**: Permite ocultar detalles internos y enfocarse en la funcionalidad principal.
- **Clases y Objetos**: Implementación de abstracción en C++; permiten modelar entidades con atributos y métodos.
- **STL**: Librería estándar que proporciona estructuras de datos optimizadas y listas para usar.

1.2.2 Declaración y Definición de Objetos

- **Declaración**: Especifica el tipo y nombre de un objeto sin asignar memoria.
- **Definición**: Asigna memoria y, opcionalmente, inicializa el objeto.
- **Ejemplo**:

```
class MiClase {
public:
    int valor;
    MiClase(int v) : valor(v) {}
};
MiClase obj(10); // Declaraci n y definici n
```

1.2.3 Arreglos de Objetos

- **Arreglos**: Colección de objetos del mismo tipo almacenados en memoria contigua.
- **Ejemplo**:

```
MiClase arr[3] = {MiClase(1), MiClase(2), MiClase(3)};
```

1.2.4 Objetos Dinámicos y Arreglos Dinámicos de Objetos

- **Objetos Dinámicos**: Se crean en tiempo de ejecución usando `new`.
- **Arreglos Dinámicos**: Se crean dinámicamente con `new[]`.
- **Ejemplo**:

```
MiClase* obj = new MiClase(10);
delete obj; // Liberar memoria

MiClase* arr = new MiClase[3];
delete[] arr; // Liberar memoria
```

1.2.5 Contenedores en STL

Contenedores Secuenciales

- **Array**: Arreglo de tamaño fijo con acceso aleatorio en $O(1)$. Ventaja: acceso rápido; Desventaja: tamaño inmutable.
- **Vector**: Arreglo dinámico; permite redimensionamiento. Inserciones al final en $O(1)$ amortizado. Ventaja: versatilidad; Desventaja: puede requerir realocaciones.
- **List**: Lista doblemente enlazada con inserciones y eliminaciones en $O(1)$. Ventaja: operaciones rápidas en cualquier posición; Desventaja: acceso aleatorio lento.
- **Forward_list**: Lista simplemente enlazada, utiliza menos memoria. Ventaja: eficiencia en memoria; Desventaja: solo permite recorrido en una dirección.
- **Deque**: Cola doble; permite inserciones y eliminaciones en ambos extremos en $O(1)$ y acceso aleatorio eficiente. Ventaja: flexibilidad; Desventaja: complejidad interna en la gestión de bloques de memoria.

Adaptadores Secuenciales

- **Stack**: Implementa una pila LIFO; se basa en contenedores subyacentes como vector o deque. Ventaja: sencillo para operaciones LIFO; Desventaja: solo permite acceso al tope.
- **Queue**: Implementa una cola FIFO; generalmente utiliza deque. Ventaja: ideal para procesamiento en orden de llegada; Desventaja: sin acceso aleatorio.
- **Priority_queue**: Cola de prioridad basada en heap binario; extrae el elemento con mayor (o menor) prioridad en $O(\log n)$. Ventaja: útil para algoritmos de planificación; Desventaja: la inserción y eliminación pueden ser costosas en comparación con una pila o cola simple.

Contenedores Asociativos

- **Ordenados**:
  - **Set**: Almacena elementos únicos ordenados. Ventaja: búsqueda, inserción y eliminación en $O(\log n)$; Desventaja: no permite duplicados.
  - **Map**: Diccionario clave-valor ordenado. Ventaja: acceso rápido a elementos por clave; Desventaja: requiere ordenamiento de claves.
  - **Multiset**: Similar a set, pero permite elementos duplicados.
  - **Multimap**: Similar a map, pero permite claves duplicadas.
- **No Ordenados**:
  - **Unordered_set**: Conjunto desordenado de elementos únicos; acceso promedio en $O(1)$. Ventaja: eficiencia; Desventaja: orden no garantizado.
  - **Unordered_map**: Diccionario clave-valor desordenado; acceso promedio en $O(1)$.
  - **Unordered_multiset** y **Unordered_multimap**: Permiten duplicados con las mismas características.

1.2.6 Énfasis en Contenedores Específicos

Listas (Homogéneas y Heterogéneas)

- **Homogéneas**: Almacenan elementos del mismo tipo.
- **Heterogéneas**: Pueden almacenar elementos de distintos tipos mediante punteros o herencia (por ejemplo, usando punteros a una clase base).

Ventajas: Flexibilidad y facilidad para insertar y eliminar elementos. **Desventajas:** Acceso aleatorio lento en comparación con arreglos.

Deque Es un contenedor que permite inserciones y eliminaciones en ambos extremos, combinando algunas ventajas de los arrays y de las listas. **Detalles:** Se implementa a través de bloques de memoria; tiene funciones `push_front`, `push_back`, `pop_front` y `pop_back`. **Casos de uso:** Ideal para implementaciones de colas dobles o buffers circulares.

Listas Enlazadas

- **Listas simplemente enlazadas**: Cada nodo tiene un puntero al siguiente. **Ventaja:** Menor uso de memoria; **Desventaja:** No permite recorrer en reversa.
- **Listas doblemente enlazadas**: Cada nodo tiene punteros al anterior y siguiente. **Ventaja:** Permiten recorrido en ambas direcciones; **Desventaja:** Mayor sobrecarga de memoria.

Colas y Colas de Prioridad

- **Queue**: Estructura FIFO, ideal para procesos en orden de llegada (por ejemplo, manejo de tareas o procesos en un sistema operativo).
- **Priority Queue**: Implementada generalmente con un heap binario; extrae el elemento con la mayor prioridad en cada operación. **Ventaja:** Eficiente para algoritmos de planificación; **Desventaja:** Las operaciones de inserción y eliminación son ligeramente más costosas ($O(\log n)$).

Pilas Implementan el modelo LIFO (último en entrar, primero en salir). Se pueden implementar usando **vector** o **list**. **Ventajas:** Acceso y eliminación en el tope en tiempo constante $O(1)$. **Casos de uso:** Algoritmos de backtracking, evaluación de expresiones, y gestión de llamadas (pila de ejecución).

1.2.7 Pro Tips en C++ y STL

- Prefiere `vector` si necesitas acceso aleatorio rápido y redimensionamiento automático.
- Utiliza `list` o `deque` si requieres frecuentes inserciones o eliminaciones en el medio o en ambos extremos.
- Para búsquedas rápidas sin necesidad de orden, `unordered_map` o `unordered_set` son ideales.
- Considera siempre el uso de iteradores para recorrer contenedores, ya que ofrecen una abstracción independiente del contenedor subyacente.

2 Árboles

2.0.1 Conceptos Básicos

- **Árbol**: Conjunto de nodos organizados jerárquicamente con una única raíz.
- **Nodos**:
  - **Raíz**: Nodo sin padre, ubicado en la cima de la jerarquía.
  - **Hoja**: Nodo sin hijos.
  - **Interno**: Nodo que tiene al menos un hijo.
  - **Nivel**: Distancia desde la raíz hasta un nodo (número de enlaces).
  - **Altura**: Longitud del camino más largo desde la raíz hasta una hoja.
- **Subárbol**: Un nodo junto con todos sus descendientes forma un subárbol.
- **Recursividad**: La estructura de un árbol se define de forma recursiva; cada subárbol es a su vez un árbol.

2.0.2 Tipos de Árboles

- **Árbol General**: Cada nodo puede tener cualquier número de hijos.
- **Árbol Binario**: Cada nodo tiene a lo sumo dos hijos.
- **Árbol Binario de Búsqueda (BST)**:
  - Los valores menores se ubican en el subárbol izquierdo y los mayores en el derecho.
  - Operaciones de búsqueda, inserción y eliminación en $O(\log n)$ en un árbol balanceado.
- **Árbol AVL**: Un BST auto-balanceado que garantiza $O(\log n)$ en todas las operaciones.
- **Árbol B**: Usado en bases de datos, permite almacenar múltiples claves por nodo y mantiene el balance.
- **Heap**: Árbol binario completo que cumple la propiedad de heap (max-heap o min-heap).

2.0.3 Recorridos de Árboles

- **Recorrido en Profundidad (DFS)**:
  - **Preorden**: Visita el nodo, luego el subárbol izquierdo y finalmente el derecho.
  - **Inorden**: Visita el subárbol izquierdo, luego el nodo, y finalmente el derecho.
  - **Postorden**: Visita primero los subárboles y luego el nodo.
- **Recorrido en Anchura (BFS)**: Se exploran los nodos nivel por nivel, utilizando una cola.

2.0.4 Implementación de Árbol Binario de Búsqueda en C++

```
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class BST {
public:
    Node* insert(Node* root, int key) {
        if (!root) return new Node(key);
        if (key < root->data)
            root->left = insert(root->left, key);
```

```
        else
            root->right = insert(root->right, key);
        return root;
    }
};
```

2.0.5 Pro Tips para Árboles

- Utiliza árboles balanceados (AVL, B) para mantener eficiencia en operaciones.
- Emplea recorridos recursivos para simplificar la lógica de visita.
- Los heaps son muy útiles para implementar colas de prioridad.

3 Grafos

3.0.1 Conceptos Básicos

- **Grafo**: Conjunto de nodos (vértices) conectados por aristas (enlaces).
- **Tipos de Grafos**:
  - **Dirigido**: Las aristas tienen dirección.
  - **No Dirigido**: Las aristas no tienen dirección.
  - **Pesado**: Cada arista tiene un peso asociado.
  - **Cíclico**: Contiene ciclos.
  - **DAG**: Grafo dirigido acíclico, ideal para ordenamientos topológicos.
  - **Conexo**: Existe un camino entre cualquier par de nodos.

3.0.2 Representaciones de Grafos

- **Lista de Adyacencia**: Cada nodo almacena una lista de vecinos. Requiere $O(|V|+|E|)$ memoria.
- **Matriz de Adyacencia**: Matriz $|V| \times |V|$ que indica conexiones. Requiere $O(|V|^2)$ memoria.
- **Lista de Arcos**: Almacena cada arista como un par (u, v). Sencilla de implementar.

3.0.3 Algoritmos en Grafos

- **Recorridos**:
  - **DFS**: Explora caminos hasta llegar al final y retrocede.
  - **BFS**: Explora los nodos nivel por nivel usando una cola.
- **Caminos más Cortos**:
  - **Dijkstra**: Para grafos con pesos no negativos ($O((V + E) \log V)$).
  - **Bellman-Ford**: Soporta pesos negativos ($O(VE)$).
  - **Floyd-Warshall**: Calcula todos los caminos más cortos ($O(V^3)$).
- **Ordenamiento Topológico**: Permite ordenar nodos de un DAG de manera que se respeten las dependencias.
- **PageRank**: Algoritmo para determinar la relevancia de páginas web en función de los enlaces entrantes.

3.0.4 Implementación de BFS en C++

```
void bfs(vector<vector<int>> &adj, int start) {
    vector<bool> visited(adj.size(), false);
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

3.0.5 Pro Tips para Grafos

- En grafos dispersos, las listas de adyacencia son más eficientes; en grafos densos, las matrices pueden ser preferibles.
- Utiliza `priority_queue` en Dijkstra para mejorar la eficiencia en la extracción del nodo de menor distancia.
- BFS es ideal para encontrar caminos mínimos en grafos no ponderados.

4 Segundo Parcial

4.1 Ordenamiento

El ordenamiento de datos es una operación fundamental en informática, ya que permite organizar la información para mejorar la búsqueda, el almacenamiento y la eficiencia de procesamiento. Dependiendo del tipo y volumen de datos, es importante elegir el algoritmo de ordenamiento adecuado.

4.1.1 Clasificación de Algoritmos de Ordenamiento

Los algoritmos de ordenamiento se pueden clasificar en tres grandes categorías:

- **Ordenamientos Simples**: Comparan elementos contiguos, requieren poca memoria y son fáciles de implementar, pero son ineficientes en grandes conjuntos de datos. Ejemplos: *Bubble Sort, Insertion Sort, Selection Sort.*
- **Ordenamientos Eficientes**: Utilizan estrategias como *divide y vencerás* para mejorar la eficiencia. Ejemplos: *Merge Sort, QuickSort.*
- **Ordenamientos Distributivos**: No dependen de comparaciones, sino que organizan los datos en estructuras intermedias. Ejemplo: *Radix Sort, Counting Sort.*

4.1.2 Ordenamientos Simples ($O(n^2)$)

Bubble Sort: Funciona comparando elementos adyacentes y permutándolos si están en el orden incorrecto. **Características**:

- Muy ineficiente para grandes volúmenes de datos.
- Requiere $O(n^2)$ comparaciones e intercambios en el peor caso.
- Se usa solo en casos educativos o listas muy pequeñas.

Insertion Sort: Inserta cada elemento en su posición correcta dentro de una lista parcialmente ordenada. **Características**:

- Es eficiente si los datos ya están casi ordenados ($O(n)$ en el mejor caso).
- Funciona bien con conjuntos pequeños.
- No es adecuado para grandes volúmenes de datos.

Selection Sort: Busca el menor elemento y lo coloca en su posición final. **Características**:

- Siempre realiza $O(n^2)$ comparaciones, pero solo $O(n)$ intercambios.
- Es ineficiente en términos de tiempo, pero eficiente en términos de intercambios.

4.1.3 Ordenamientos Eficientes ($O(n \log n)$)

Merge Sort: Utiliza la estrategia de dividir y conquistar, separando el arreglo en mitades, ordenándolas y combinándolas. **Características**:

- Siempre garantiza $O(n \log n)$ comparaciones.
- Utiliza más memoria debido a la recursión y listas auxiliares.
- Se usa cuando se requiere estabilidad en el ordenamiento.

QuickSort: Elige un pivote, divide los datos en menores y mayores al pivote, y aplica recursión. **Características**:

- Tiene un mejor rendimiento en la práctica comparado con Merge Sort.
- En promedio, opera en $O(n \log n)$, pero en el peor caso puede ser $O(n^2)$ si el pivote no se elige bien.
- Se utiliza ampliamente por su velocidad y baja necesidad de memoria adicional.

4.2 Búsqueda

La búsqueda es otra operación fundamental que permite localizar elementos en estructuras de datos. Dependiendo del tamaño y organización de los datos, ciertos algoritmos son más eficientes que otros.

4.2.1 Búsqueda Binaria ($O(\log n)$)

Descripción:

- Se aplica solo en listas ordenadas.
- Divide la lista en dos en cada paso, descartando la mitad donde no está el elemento buscado.
- En el mejor caso ($O(1)$), encuentra el elemento en la primera comparación.
- En el peor caso ($O(\log n)$), realiza múltiples divisiones hasta encontrarlo o descartar su existencia.

4.2.2 Búsqueda por Interpolación

Descripción:

- Similar a la búsqueda binaria, pero en lugar de dividir en el punto medio, usa una fórmula para estimar la posición del elemento buscado.
- Funciona bien cuando los datos están distribuidos uniformemente.
- En el mejor caso ($O(1)$), encuentra el elemento en un solo paso.
- En el peor caso ($O(n)$), se degrada si los datos están desordenados o la estimación es inexacta.

4.3 Hashing

El hashing es una técnica que permite acceder a los datos en tiempo constante $O(1)$ en promedio, utilizando una función hash para transformar las claves en posiciones dentro de una tabla hash.

4.3.1 Conceptos Clave

- **Función Hash**: Convierte una clave en un índice en la tabla.
- **Colisiones**: Ocurren cuando dos claves diferentes generan el mismo índice.
- **Técnicas de Resolución de Colisiones**:
  - **Hashing Abierto**: Usa estructuras externas como listas enlazadas.
  - **Hashing Cerrado**: Encuentra otra posición en la tabla mediante sondeo lineal, cuadrático o doble hashing.

4.3.2 Función Hash DJB2

Es una función hash eficiente y sencilla, diseñada para distribuir valores uniformemente. **Características**:

- Utiliza una multiplicación constante y sumas sucesivas.
- Es rápida y fácil de implementar.
- Se usa ampliamente en la práctica para claves alfanuméricas.

4.3.3 Rehashing

- Cuando la tabla hash se llena, se duplica su tamaño y se recalculan los índices.
- Es necesario para mantener la eficiencia de acceso $O(1)$.
- Implica un costo adicional al momento de redimensionar la tabla.

4.4 Pro Tips

- **Ordenamiento:** Preferir QuickSort sobre Bubble Sort para grandes conjuntos de datos.
- **Búsqueda:** Utilizar Búsqueda Binaria solo en listas ordenadas.
- **Hashing:** Para claves dispersas, las tablas hash son más eficientes que los árboles binarios.
- **Factor de Carga:** Mantenerlo entre 0.6 y 0.75 en tablas hash para evitar colisiones.
- **Elección de Algoritmos:** Merge Sort es mejor si se necesita estabilidad en el ordenamiento.

5 Estrategias Algorítmicas

En la práctica, muchas veces necesitamos diseñar nuestros propios algoritmos para resolver problemas específicos. Para ello, podemos aprovechar distintas estrategias algorítmicas que optimizan el rendimiento y la eficiencia.

5.1 Búsqueda por Fuerza Bruta

La estrategia de fuerza bruta consiste en evaluar todas las combinaciones posibles hasta encontrar la solución óptima.

5.1.1 Ventajas y Desventajas

- **Ventajas**:
  - Simple de implementar.
  - Funciona para cualquier problema.
- **Desventajas**:
  - Ineficiente en problemas con gran cantidad de combinaciones (explosión combinatoria).
  - Puede ser impráctico en términos de tiempo y memoria.

5.1.2 Ejemplo: Encontrar Divisores de un Número

Para hallar los divisores de un número n, probamos todos los valores de 1 a n.

```
void encontrarDivisores(int n) {
    for (int i = 1; i <= n; i++) {
        if (n % i == 0)
            cout << i << "-";
    }
}
```

5.1.3 Optimización con Paralelismo

Podemos distribuir el cálculo en múltiples procesadores:

- Una CPU moderna tiene múltiples núcleos que pueden dividir el trabajo.
- Una GPU puede ejecutar miles de operaciones simultáneamente.
- Se pueden distribuir tareas en una red de computadoras.

5.2 Divide and Conquer

La estrategia divide-and-conquer divide un problema en subproblemas más pequeños, los resuelve de forma recursiva y luego combina los resultados.

5.2.1 Pasos Clave

1. Dividir el problema en subproblemas más pequeños.
2. Resolver cada subproblema recursivamente.
3. Combinar las soluciones para obtener el resultado final.

5.2.2 Ejemplo: Merge Sort

```
void mergeSort(vector<int> &arr, int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

5.2.3 Optimización: Multiplicación de Matrices

El método clásico de multiplicación de matrices es $O(n^3)$. Con divide-and-conquer, podemos mejorar la eficiencia.

- Dividimos las matrices en submatrices más pequeñas.
- Aplicamos la multiplicación recursivamente.
- Combinamos los resultados usando sumas y productos de submatrices.

Complejidad Reducida: De $O(n^3)$ a $O(n^2)$ en algunos casos.

5.3 Programación Dinámica

La programación dinámica es una técnica que divide un problema en subproblemas solapados, evitando cálculos repetitivos mediante almacenamiento en memoria.

5.3.1 Diferencia con Divide-and-Conquer

Mientras que divide-and-conquer resuelve subproblemas independientes, la programación dinámica evita cálculos repetidos mediante una técnica llamada **memoización**.

5.3.2 Ejemplo: Corte de Barras

Dado un precio por cada longitud de barra, buscamos la mejor forma de cortar una barra de longitud n para maximizar la ganancia.

```
int cutRod(vector<int> &prices, int n) {
    if (n == 0) return 0;
    int maxRevenue = -INFINITY;
    for (int i = 1; i <= n; i++) {
        maxRevenue = max(maxRevenue, prices[i] + cutRod(prices, n - i));
    }
    return maxRevenue;
}
```

5.3.3 Optimización con Memoización

Podemos mejorar la eficiencia almacenando los valores ya calculados.

```
vector<int> memo(n + 1, -1);
int cutRodMemo(vector<int> &prices, int n) {
    if (n == 0) return 0;
    if (memo[n] != -1) return memo[n];
    int maxRevenue = -INFINITY;
    for (int i = 1; i <= n; i++) {
        maxRevenue = max(maxRevenue, prices[i] + cutRodMemo(prices, n - i));
    }
    return memo[n] = maxRevenue;
}
```

5.3.4 Ventajas y Desventajas

- **Ventajas**:
  - Reduce la complejidad de muchos problemas de $O(2^n)$ a $O(n^2)$ o $O(n)$.
  - Evita cálculos innecesarios.
- **Desventajas**:
  - Puede requerir más memoria para almacenar resultados.
  - No es aplicable si los subproblemas no se solapan.

5.4 Pro Tips Generales

- **Usar fuerza bruta solo en problemas pequeños**, ya que su costo crece exponencialmente.
- **Divide-and-conquer es ideal para problemas recursivos**, pero debemos manejar correctamente la recursión para evitar desbordamiento de pila.
- **La programación dinámica es clave en optimización**, pero debemos identificar correctamente los subproblemas solapados.
- **Cuando sea posible, combinar estrategias**. Ejemplo: Dijkstra (algoritmo goloso) puede mejorarse con programación dinámica en algunos casos.