Documentación Seguimiento 4 Discretas

- Nicolas buelvas
- Daniel Nastul
- Sebastian

Tabla de Ejecución (Main)

Línea de Código	Complejidad Temporal	Complejidad Espacial	Comentarios
package ui;	O(1)	O(1)	Declaración del paquete, sin impacto en la ejecución.
import java.util.Scanner;	O(1)	O(1)	Importación necesaria para entrada de datos.
import model.Controller;	O(1)	O(1)	Importación de la clase Controller, única interacción con model.
public class Main {	O(1)	O(1)	Declaración de la clase.
public static void main(String[] args) {	O(1)	O(1)	Inicio del método principal.
Scanner sc = new Scanner(System.in);	O(1)	O(1)	Creación del objeto Scanner para lectura; espacio adicional constante.
System.out.println("Ingrese la cantidad de elementos del arreglo:");	O(1)	O(1)	Impresión de mensaje, operación constante.
int n = sc.nextInt();	O(1)	O(1)	Lectura de un entero.
int[] arr = new int[n];	O(1)	O(n)	Reserva memoria para un arreglo de n elementos.
System.out.println("Ingrese los elementos del arreglo:");	O(1)	O(1)	Impresión de mensaje.
for (int i = 0; i < n; i++) {	O(n)	O(1)	Bucle que itera n veces para leer cada elemento.

arr[i] = sc.nextInt();	O(1) por iteración	O(1)	Cada lectura es O(1); en total O(n)
}	O(n)	O(1)	Fin del bucle.
sr.nextLine();	O(1)	O(1)	Limpiar el buffer
System.out.println("Ingrese la suma objetivo S:");	O(1)	O(1)	Impresión de mensaje.
int S = sc.nextInt();	O(1)	O(1)	Lectura del valor de S.
Controller controller = new Controller();	O(1)	O(1)	Instanciación del Controller; costo constante.
String resultado = controller.encontrarSubarreglo(arr, S);	O(n)*	O(1)*	Llama al método que ejecuta la lógica (ver análisis de SumaSubarreglo); en promedio O(n).
System.out.println(resultado);	O(1)	O(1)	Impresión del resultado recibido.
sc.close();	O(1)	O(1)	Cierre del Scanner, operación constante.
}	O(1)	O(1)	Fin del método main.
}	O(1)	O(1)	Fin de la clase.

Análisis de Complejidad Temporal y Espacial

Complejidad por Clase

Clase Main

- **Complejidad Temporal:** O(n)
  - Se recorre el arreglo para la lectura de datos (O(n)).
  - Se llama a encontrarSubarreglo que tiene una complejidad O(n).
  - En total, la complejidad es **O(n) + O(n) = O(n)**.
- **Complejidad Espacial:** O(n)
  - Se almacena un arreglo de tamaño n.
  - La memoria usada es proporcional a la entrada, por lo que es **O(n)**.

Tabla de Ejecución (Controller)

Línea de Código	Complejidad Temporal	Complejidad Espacial	Comentarios
package model;	O(1)	O(1)	Declaración del paquete.
import structure.SumaSubarreglo;	O(1)	O(1)	Importación de la clase que contiene la lógica del subarreglo.
public class Controller {	O(1)	O(1)	Declaración de la clase.
public String encontrarSubarreglo(int[] arr, int S) {	O(1)	O(1)	Inicio del método; preparación para delegar la búsqueda a SumaSubarreglo.
SumaSubarreglo sumaSubarreglo = new SumaSubarreglo();	O(1)	O(1)	Instancia de SumaSubarreglo; costo constante.
int[] indices = sumaSubarreglo.encontrarSubarreglo(arr, S);	O(n)*	O(1)*	Llama a la lógica de negocio; se analizará en SumaSubarreglo (en promedio O(n) en tiempo).
if (indices[0] == -1) {	O(1)	O(1)	Evaluación condicional.
return "No se encontró un subarreglo con la suma " + S;	O(1)	O(1)	Retorno de mensaje; concatenación de cadenas de longitud constante.
} else {	O(1)	O(1)	Alternativa en caso de hallazgo.
return "Subarreglo encontrado: (" + indices[0] + ", " + indices[1] + ")";	O(1)	O(1)	Retorno de mensaje con los índices encontrados.
}	O(1)	O(1)	Fin del condicional.

}	O(1)	O(1)	Fin del método.
}	O(1)	O(1)	Fin de la clase.

Clase Controller

- **Complejidad Temporal:** O(n)
  - Llama a encontrarSubarreglo de SumaSubarreglo, que es O(n).
  - El resto de las operaciones son O(1), por lo que la complejidad sigue siendo **O(n)**.
- **Complejidad Espacial:** O(1)
  - Solo almacena un objeto de SumaSubarreglo y variables auxiliares.
  - La memoria usada no crece con la entrada, por lo que es **O(1)**.

Tabla de Ejecución (SumaSubarreglo)

Línea de Código	Complejidad Temporal	Complejidad Espacial	Comentarios
package structure;	O(1)	O(1)	Declaración del paquete.
public class SumaSubarreglo {	O(1)	O(1)	Declaración de la clase.
public int[] encontrarSubarreglo(int[] arr, int S) {	O(1)	O(1)	Inicio del método.
TablaHash hash = new TablaHash(arr.length * 2);	O(n)*	O(n)*	Instancia de TablaHash: crea un arreglo interno de tamaño ~2n; se analiza en TablaHash.
int sumaAcumulada = 0;	O(1)	O(1)	Inicialización de la suma acumulada.
for (int i = 0; i < arr.length; i++) {	O(n)	O(1)	Bucle principal que itera sobre cada elemento del arreglo.
sumaAcumulada += arr[i];	O(1) por iteración	O(1)	Operación de suma, se ejecuta n veces.
if (sumaAcumulada == S) {	O(1)	O(1)	Verifica si la suma acumulada coincide con S.
return new int[]{0, i};	O(1)	O(1)	Retorno inmediato en caso afirmativo.
}	O(1)	O(1)	Fin de la condición.
Integer indiceAnterior = hash.get(sumaAcumulada - S);	O(1) (promedio)	O(1)	Consulta en la tabla hash; operación promedio constante.

if (indiceAnterior != null) {	O(1)	O(1)	Evaluación condicional
return new int[] {indiceAnterior + 1, i};	O(1)	O(1)	Retorno inmediato si se encuentra la suma complementaria.
}	O(1)	O(1)	Fin del condicional.
hash.put(sumaAcumulada, i);	O(1) (promedio)	O(1)	Inserta la suma acumulada y el índice en la tabla hash; operación promedio constante.
}	O(n)	O(1)	Fin del bucle principal.
return new int[]{-1, -1};	O(1)	O(1)	Retorno si no se encuentra ningún subarreglo que cumpla la condición.
}	O(1)	O(1)	Fin del método.
}	O(1)	O(1)	Fin de la clase.

Clase SumaSubarreglo

- **Complejidad Temporal:** O(n)
  - Se recorre el arreglo una vez (O(n)).
  - Las operaciones de la tabla hash (inserción y búsqueda) son O(1) en promedio.
  - En total, la complejidad es **O(n) + O(1) = O(n)**.
- **Complejidad Espacial:** O(n)
  - Se usa una tabla hash que en el peor caso puede almacenar n elementos.
  - En total, la memoria usada es **O(n)**.

Tabla de Ejecución (TablaHash)

Línea de Código	Complejidad Temporal	Complejidad Espacial	Comentarios
package structure;	O(1)	O(1)	Declaración del paquete.
public class TablaHash {	O(1)	O(1)	Declaración de la clase.
private ListaEnlazada[] tabla;	O(1)	O(1)	Declaración de variable miembro.
private int capacidad;	O(1)	O(1)	Declaración de variable miembro.
public TablaHash(int capacidad) {	O(1)	O(1)	Inicio del constructor.
this.capacidad = capacidad;	O(1)	O(1)	Asignación, constante.
tabla = new ListaEnlazada[capacidad];	O(1)	O(capacidad)	Reserva un arreglo de objetos; espacio proporcional a la capacidad (O(n) en función de n).
for (int i = 0; i < capacidad; i++) {	O(capacidad)	O(1)	Bucle para inicializar cada posición del arreglo.
tabla[i] = new ListaEnlazada();	O(1) por iteración	O(1)	Cada iteración crea una lista enlazada; en total O(capacidad) operaciones.
}	O(capacidad)	O(1)	Fin del bucle.
}	O(capacidad)	O(capacidad)	Fin del constructor; en tiempo y espacio, O(n) con n = capacidad.
private int hash(int key) {	O(1)	O(1)	Función hash simple, operación constante.

return Math.abs(key) % capacidad;	O(1)	O(1)	Cálculo del índice hash.
}	O(1)	O(1)	Fin del método hash.
public void put(int key, int value) {	O(1) (promedio)	O(1)	Inserción en la tabla hash; se asume costo promedio constante.
int indice = hash(key);	O(1)	O(1)	Cálculo del índice en la tabla.
tabla[indice].agregar(key, value);	O(1)	O(1)	Llama a agregar de ListaEnlazada, que es O(1).
}	O(1)	O(1)	Fin del método put.
public Integer get(int key) {	O(1) (promedio)	O(1)	Consulta en la tabla hash; costo promedio constante.
int indice = hash(key);	O(1)	O(1)	Cálculo del índice.
return tabla[indice].obtenerValor(key);	O(1) (promedio)	O(1)	Llama a obtenerValor de ListaEnlazada; en promedio O(1).
}	O(1)	O(1)	Fin del método get.
}	O(1)	O(1)	Fin de la clase.

Clase TablaHash

- **Complejidad Temporal:** O(1) en promedio, O(n) en el peor caso
  - **Inserción:** O(1) en promedio, O(n) en el peor caso (colisiones excesivas).
  - **Búsqueda:** O(1) en promedio, O(n) en el peor caso.
  - En el caso promedio, las operaciones son **O(1)**.
- **Complejidad Espacial:** O(n)
  - Depende del número de elementos almacenados en la tabla.
  - En total, la memoria usada es **O(n)**.

Tabla de Ejecución (ListaEnlazada)

Línea de Código	Complejidad Temporal	Complejidad Espacial	Comentarios
package structure;	O(1)	O(1)	Declaración del paquete.
public class ListaEnlazada {	O(1)	O(1)	Declaración de la clase.
private Nodo head;	O(1)	O(1)	Declaración de variable miembro.
public ListaEnlazada() {	O(1)	O(1)	Constructor; inicialización de la lista.
head = null;	O(1)	O(1)	Asigna null al head, constante.
}	O(1)	O(1)	Fin del constructor.
public void agregar(int key, int value) {	O(1)	O(1)	Método para agregar un nodo; inserción al inicio, costo constante.
Nodo nuevo = new Nodo(key, value);	O(1)	O(1)	Creación del nuevo nodo.
nuevo.next = head;	O(1)	O(1)	Enlaza el nuevo nodo con el nodo actual head.
head = nuevo;	O(1)	O(1)	Actualiza head; operación constante.
}	O(1)	O(1)	Fin del método agregar.
public Integer obtenerValor(int key) {	O(1)	O(1)	Inicio del método; búsqueda lineal en la lista.
Nodo actual = head;	O(1)	O(1)	Inicializa la variable para recorrer la lista.

while (actual != null) {	O(k)	O(1)	En el peor caso recorre k nodos; k pequeño en promedio si la tabla hash está bien distribuida.
if (actual.key == key) {	O(1)	O(1)	Comparación constante por nodo.
return actual.value;	O(1)	O(1)	Retorna el valor encontrado.
}	O(1)	O(1)	Fin de la condición.
actual = actual.next;	O(1)	O(1)	Avanza al siguiente nodo.
}	O(k)	O(1)	Fin del bucle while; en peor caso, k iteraciones.
return null;	O(1)	O(1)	Retorna null si no se encuentra la clave.
}	O(1)	O(1)	Fin del método obtenerValor.
}	O(1)	O(1)	Fin de la clase.

Clase ListaEnlazada

- **Complejidad Temporal:**
  - **Inserción:** O(1) (se agrega al inicio de la lista).
  - **Búsqueda:** O(n) en el peor caso (recorrer toda la lista).
- **Complejidad Espacial:** O(n)
  - Se almacena cada nodo con su clave y valor.
  - En total, la memoria usada es **O(n)**.

Tabla de Ejecución (Nodo)

Línea de Código	Complejidad Temporal	Complejidad Espacial	Comentarios
package structure;	O(1)	O(1)	Declaración del paquete.
public class Nodo {	O(1)	O(1)	Declaración de la clase.
public int key;	O(1)	O(1)	Declaración de variable miembro.
public int value;	O(1)	O(1)	Declaración de variable miembro.
public Nodo next;	O(1)	O(1)	Declaración de referencia al siguiente nodo.
public Nodo(int key, int value) {	O(1)	O(1)	Constructor, inicialización de variables.
this.key = key;	O(1)	O(1)	Asignación constante.
this.value = value;	O(1)	O(1)	Asignación constante.
this.next = null;	O(1)	O(1)	Inicializa la referencia next en null.
}	O(1)	O(1)	Fin del constructor.
}	O(1)	O(1)	Fin de la clase.

Clase Nodo

- **Complejidad Temporal:** O(1)
  - Creación de un nodo toma tiempo constante.
- **Complejidad Espacial:** O(1)
  - Cada nodo ocupa espacio fijo en memoria.

Análisis Global

- **Complejidad Temporal Total:** O(n) en el caso promedio
  - El flujo principal involucra recorrer el arreglo (O(n)) y realizar operaciones en la tabla hash (O(1) en promedio).

    - En el peor caso (colisiones en la tabla hash), las operaciones de búsqueda pueden volverse O(n), resultando en una complejidad total O(n^2).
- **Complejidad Espacial Total:** O(n)
    - El principal consumo de memoria proviene del almacenamiento del arreglo y la tabla hash.
    - En el peor caso, se almacena una cantidad proporcional a n, resultando en O(n).

Conclusión: El algoritmo es eficiente en la mayoría de los casos con **O(n) de tiempo promedio y O(n) de espacio**. Sin embargo, en situaciones con muchas colisiones en la tabla hash, el tiempo podría degradarse a **O(n^2)**. La estructura de datos utilizada permite una buena eficiencia en la resolución del problema.