

FASE 1: COMPRENDE

1.- Visualizador de Memoria

- Diagrama propio que muestre la fórmula de acceso aplicada a un ejemplo diferente

Dirección	Cálculo	Valor
1000	$d_{base} + (0 \times 4) = 1000$	[5] (arreglo 1)
1004	$d_{base} + (1 \times 4) = 1004$	[10] (arreglo 2)
1008	$d_{base} + (2 \times 4) = 1008$	[15] (arreglo 3)
1012	$d_{base} + (3 \times 4) = 1012$	[20] (arreglo 4)
1016	$d_{base} + (4 \times 4) = 1016$	[25] (arreglo 5)
1020	$d_{base} + (5 \times 4) = 1020$	[30] (arreglo 6)

- tabla comparativa: operación vs. número de movimientos necesarios para un arreglo de n=10

operación	posición	m necesarios
acceso	9	0
inserción	8	1
eliminación	7	2

nota: use posiciones dentro del rango de n=10

- Responde: ¿Por qué insertar al inicio es más costoso que insertar al final?

ya que al insertar al inicio tienes que recorrer todos los arreglos mientras que si insertas al final no se recorre ningun arreglo

2.- Detective del Caso MegaStore

- Fórmula matemática del costo de n inserciones con incremento de +1
 $n(n-1)/2$
- Fórmula del costo con duplicación de capacidad
 $2\lceil \log_2 n \rceil - 1$ (los corchetes redondean el resultado de lo que hay dentro)
- Informe breve (150 palabras) explicando a un gerente no técnico por qué el sistema era lento

el programa que teníamos antes lo que hacía era que cuando un dato nuevo era añadido, este creaba otro conjunto de datos donde el conjunto de datos anterior pasaba todos los datos que tenía y agregaba el nuevo producto, esto aplicándolo a una analogía seria que cuando alguien se queda sin hojas en una libreta, consigue una libreta con una hoja más que la anterior y pasa toda la información a la nueva libreta, lo que hace lento el manejo de la información

Checkpoint Metacognitivo - Fase COMPRENDE

- ¿Puedo explicar con mis propias palabras por qué $\text{arr}[i]$ es $O(1)$ usando la fórmula de dirección?

si, no importa qué posición sea la del arreglo, siempre va a hacer el mismo recorrido

- ¿Qué imagen mental tengo ahora de un arreglo en memoria? ¿Es diferente a lo que pensaba antes?

no es tan diferente, a final de cuentas un arreglo es como una caja donde se pueden guardar valores de un solo mismo tipo, nada mas que ahora se que hay arreglos dinamicos y estaticos

- Si me preguntaran "¿por qué insertar al inicio es $O(n)$? ", ¿podría dibujarlo paso a paso?

dibujarlo no estoy seguro pero insertar al inicio es lineal debido a que el numero de movimientos crece linealmente con n, osea que si quieres introducir un valor al inicio todos los valores se desplazaran una vez

- ¿Qué conexión veo entre el análisis de complejidad de la Semana 1 y las operaciones de arreglos?
que dependiendo de como estructuras tu código puede comportarse de distintas formas y tienes que buscar en qué complejidad encaja

FASE 2: APLICA

3.- Arquitecto de Código

- Código completo del TAD ArregloDinámico (pseudocódigo o lenguaje real)

```
#include <iostream>
#include <stdexcept> // Para manejar excepciones como
std::out_of_range

class ArregloDinamico {
private:
    int* elementos; // Puntero dinámico que apunta al arreglo en memoria
    int capacidad; // Capacidad total del arreglo dinámico
    int tamano; // Número actual de elementos en el arreglo

    // Método privado para redimensionar (duplicar) la capacidad del arreglo
    void redimensionar() {
        int* nuevoBloque = new int[capacidad * 2]; // Duplicar la capacidad
        for (int i = 0; i < tamano; i++) {
            nuevoBloque[i] = elementos[i]; // Copiar los elementos existentes
        }
        delete[] elementos; // Liberar la memoria del arreglo original
        elementos = nuevoBloque; // Actualizar el puntero al nuevo arreglo
        capacidad *= 2; // Actualizar la capacidad
    }
}
```

```

public:
    // Constructor: Inicializa los atributos
    ArregloDinamico() {
        capacidad = 2; // Capacidad inicial del arreglo dinámico
        tamano = 0; // Comienza vacío
        elementos = new int[capacidad]; // Arreglo dinámico inicial
    }

    // Destructor: Libera la memoria dinámica utilizada por el arreglo
    ~ArregloDinamico() {
        delete[] elementos; // Liberar la memoria cuando el objeto sea
        destruido
    }

    // Método para agregar un elemento al final del arreglo
    void agregar(int elemento) {
        if (tamano == capacidad) { // Si el arreglo está lleno
            redimensionar(); // Redimensionar (duplicar capacidad)
        }
        elementos[tamano] = elemento; // Agregar el elemento al final
        tamano++; // Incrementar el tamaño
    }

    // Método para insertar un elemento en una posición específica
    void insertar(int indice, int elemento) {
        if (indice < 0 || indice > tamano) { // Índice fuera de rango
            throw std::out_of_range("Índice inválido");
        }
        if (tamano == capacidad) { // Si el arreglo está lleno
            redimensionar(); // Redimensionar el arreglo
        }
        for (int i = tamano - 1; i >= indice; i--) { // Mover a la derecha
            elementos[i + 1] = elementos[i];
        }
        elementos[indice] = elemento; // Insertar el nuevo elemento
        tamano++; // Incrementar el tamaño
    }

    // Método para eliminar un elemento en una posición específica
    void eliminar(int indice) {
        if (indice < 0 || indice >= tamano) { // Índice fuera de rango
            throw std::out_of_range("Índice inválido");
        }
        for (int i = indice; i < tamano - 1; i++) { // Mover a la izquierda
            elementos[i] = elementos[i + 1];
        }
        tamano--; // Incrementar el tamaño
    }

```

```

        elementos[i] = elementos[i + 1];
    }
    tamano--; // Reducir el tamaño del arreglo

    // Opcional: Reducir la capacidad si se usa menos de la mitad
    if (tamano < capacidad / 2) {
        int nuevaCapacidad = capacidad / 2;
        int* nuevoBloque = new int[nuevaCapacidad];
        for (int i = 0; i < tamano; i++) {
            nuevoBloque[i] = elementos[i];
        }
        delete[] elementos; // Liberar la memoria antigua
        elementos = nuevoBloque; // Actualizar el puntero
        capacidad = nuevaCapacidad; // Actualizar la capacidad
    }
}

// Método para imprimir los elementos del arreglo (para pruebas)
void imprimir() const {
    for (int i = 0; i < tamano; i++) {
        std::cout << elementos[i] << " ";
    }
    std::cout << std::endl;
}

// Método para obtener el tamaño actual
int obtenerTamano() const {
    return tamano;
}

// Método para obtener la capacidad actual
int obtenerCapacidad() const {
    return capacidad;
}
};

```

- Tabla de complejidades: cada método con su $O()$ temporal y espacial

método	Complejidad Temporal (O)	Complejidad Espacial (O)	Explicación
Constructor	O(1)	O(n)	Inicializa el arreglo con capacidad inicial. La memoria dinámica utilizada es proporcional a la capacidad inicial (n).
Destructor	O(1)	O(1)	Solo libera la memoria dinámica utilizada por el puntero, no requiere operaciones adicionales.
agregar(elemento)	O(1) (amortizada), O(n) (en redimensionar)	O(1)	Agregar es O(1) si hay capacidad suficiente, pero O(n) si hay que redimensionar y copiar los elementos al nuevo arreglo dinámico.
redimensionar	O(n)	O(n)	Duplica la capacidad del arreglo y copia todos los elementos al nuevo arreglo, utilizando memoria intermedia temporal.
insertar(indice, elemento)	O(n)	O(1)	Desplaza los elementos desde el índice especificado hacia la derecha y luego inserta el nuevo elemento allí.
eliminar(indice)	O(n)	O(n)	Desplaza los elementos hacia la izquierda para llenar el hueco. Puede requerir redimensionamiento hacia abajo en algunos casos.
imprimir()	O(n)	O(1)	Recorre el arreglo y muestra todos los elementos actuales en una sola operación lineal.
obtenerTamano()	O(1)	O(1)	Devuelve el número actual de elementos almacenados en el arreglo (un simple acceso a la variable interna).
obtenerCapacidad()	O(1)	O(1)	Devuelve la capacidad actual del arreglo (un simple acceso a la variable interna).

- Lista de al menos 3 casos borde que tu implementación maneja
 - 1.- insertar o eliminar usando un índice inválido
 - 2.- capacidad insuficiente al agregar o insertar elementos
 - 3.- redimensionamiento a la mitad cuando el arreglo está subutilizado

4.- Laboratorio de Problemas Clásicos

- Pseudocódigo de tu solución para cada problema
problema 1 rotar un arreglo:

Entrada: arreglo arr de tamaño n, número k

Salida: arreglo arr rotado k posiciones a la derecha in-place

$k \leftarrow k \bmod n$

Llamar Revertir(arr, 0, n - 1)

Llamar Revertir(arr, 0, n - k - 1)

Llamar Revertir(arr, n - k, n - 1)

FinAlgoritmo

Procedimiento Revertir(arr, inicio, fin)

Mientras inicio < fin Hacer

 temp \leftarrow arr[inicio]

 arr[inicio] \leftarrow arr[fin]

 arr[fin] \leftarrow temp

 inicio \leftarrow inicio + 1

 fin \leftarrow fin - 1

FinMientras

FinProcedimiento

Problema 2 eliminar duplicados:

Entrada: arreglo nums de tamaño n
Salida: Longitud del arreglo modificado con números únicos

Si $n = 0$ entonces

 Retornar 0

 Inicializar $i \leftarrow 0$

 Para $j \leftarrow 1$ Hasta $n-1$ Hacer

 Si $\text{nums}[j] \neq \text{nums}[i]$ Entonces

$i \leftarrow i + 1$

$\text{nums}[i] \leftarrow \text{nums}[j]$

 FinSi

 FinPara

 Retornar $i + 1$

FinAlgoritmo

Problema 3 mover ceros al final:

Entrada: arreglo nums con n elementos

Salida: el arreglo nums con ceros movidos al final

 Inicializar lastNonZeroIndex $\leftarrow 0$

 Para $i \leftarrow 0$ Hasta $n-1$ Hacer

 Si $\text{nums}[i] \neq 0$ Entonces

 Intercambiar $\text{nums}[\text{lastNonZeroIndex}] \leftrightarrow \text{nums}[i]$

$\text{lastNonZeroIndex} \leftarrow \text{lastNonZeroIndex} + 1$

 FinSi

 FinPara

FinAlgoritmo

Problema 4 encontrar elemento mayoritario:

Entrada: arreglo nums de tamaño n

Salida: elemento mayoritario

 Inicializar candidate \leftarrow Ninguno

 Inicializar count $\leftarrow 0$

 Para cada num en nums Hacer

 Si count = 0 Entonces

 candidate \leftarrow num

FinSi

Si num = candidate Entonces

 count ← count + 1

Sino

 count ← count - 1

FinSi

FinPara

Retornar candidate

FinAlgoritmo

- Análisis de complejidad temporal y espacial de cada solución

Solución 1 (revertir el arreglo):

Complejidad temporal:

Cada operación de revertir tiene una complejidad de $O(n)$

Complejidad espacial:

Solo usamos memoria adicional para variables auxiliares como temp en el proceso de intercambio, por lo que la complejidad es $O(1)$

Solución 2 (eliminar duplicados):

Complejidad temporal:

Recorremos el arreglo una sola vez con el bucle, por lo que la complejidad es $O(n)$

Complejidad espacial:

No hay estructuras auxiliares, y todo se realiza in-place

Complejidad espacial: ($O(1)$)

Solución 3 (mover ceros al final):

Complejidad temporal:

Igual que en la anterior solución solo recorremos el arreglo una vez con el bucle, por lo que la complejidad es $O(n)$

Complejidad espacial:

No usamos memoria auxiliar para almacenar los valores por lo que la complejidad sigue siendo $O(1)$

Solución 4 (encontrar el elemento mayoritario):

Complejidad temporal:

igual que en los otros casos solo se recorre una vez el bucle, por lo que sigue siendo $O(n)$

Complejidad espacial:

Solo usamos unas pocas variables auxiliares (candidate y count), por lo que la complejidad es $O(1)$

- Para al menos un problema, muestra la evolución de tu solución si la mejoraste
en el problema uno la solución que termine implementando fue revertir el arreglo, de esta forma me ahorró el hecho de usar un arreglo auxiliar, así pudiendo ahorrar memoria

Checkpoint Metacognitivo - Fase APLICA

- ¿Puedo implementar `agregar()` sin mirar mis notas? ¿Qué parte me cuesta más recordar?
no creo poder pero puedo hacer el intento, lo que mas me cuesta recordar es como funcionan los punteros, se que hacen de forma teórica pero por alguna razón memorizar como se aplican de forma práctica me sigue costando algo de trabajo
- Cuando la IA me hizo preguntas socráticas, ¿en qué momento me di cuenta de algo que no había considerado?
cuando de plano no sabía que responderle
- De los 4 problemas, ¿cuál me costó más trabajo? ¿Qué patrón o técnica me faltaba?
definitivamente todos me costaron mucho trabajo, debido que para cumplir todos los requisitos que pedían los problemas tenía que implementar cosas que pasé muy por encima el semestre pasado, así que si bien en esto sufri demasiado, lo visto en esta fase solo me motiva a estudiar más respecto a la programación de códigos
- ¿Cómo cambió mi aproximación a los problemas entre el primero y el último?
en casi nada, sufri en todos

FASE 3: REFLEXIONA

5.- Diálogo Socrático sobre Decisiones de Diseño

- Documento de reflexión con: cada decisión de diseño, la justificación refinada después del diálogo, y un escenario donde la cambiarías

decisión 1:

1.- ¿Cuál es el patrón de tamaños esperado de tus arreglos (distribución aproximada: 0–3, 4–10, 11–100, 1000+)?

el patrón esperado de mis arreglos es de 4-10

2.- ¿Cuántas instancias típicamente existen al mismo tiempo (decenas, miles, millones)?

las instancias que existen al mismo tiempo son decenas o menos ya que estoy trabajando con un arreglo pequeño

3.- ¿Qué te preocupa más: memoria promedio, picos de latencia por resize, o simplicidad?

ahorita por lo mismo de que apenas soy nuevo en el tema me preocupa la simplicidad de mi trabajo

decisión 2:

1.- ¿Cuál crees que es el caso más común de crecimiento en tus arreglos?

mis arreglos crecen lentamente debido a que son pequeños

2.- Dado que estás optimizando para simplicidad ahora:

- **¿Sientes que 2x es suficientemente intuitivo y razonable para ti?**
- **¿Hay algún caso donde quisieras desviarte del 2x por algo más adaptado a tu dominio?**

de momento el 2x si se me hace razonable y intuitivo de usar debido a que como mi capacidad inicial es 10, al usar el 2x no siento que este utilizando muchos recursos

decisión 3:

1.- ¿Por qué elegiste no implementar reducción de capacidad?

respecto a la primera pregunta no quise implementar reducción de capacidad por que como solo es un ejercicio práctico y sencillo realmente no lo considere necesario, haciendo la implementación más simple y directa

2.- ¿Qué sucede a largo plazo si un arreglo crece mucho y luego se libera la mayoría de sus elementos?

respecto a la segunda pregunta no sé qué pase pero asumo que de alguna forma habrá problemas de la gestión de memoria, pero como dije lo que estoy haciendo es muy simple así que no me termino de preocupar

3.- ¿En qué escenario sería un problema depender solo de “crecimiento”?

y respecto a la tercera pregunta considero que sería un problema solo depender de la latencia sería cuando tenemos un arreglo inicial pequeño como es mi caso, pero como dije estoy trabajando con arreglos pequeños así que eso no me importa mucho

decisión 4:

1.- ¿Por qué decidiste lanzar excepciones?

creo que las excepciones son mejores que devolver un valor especial indicativo porque así puedo indicar que hay un error

2.- ¿Qué tipo de excepción lanzas y por qué?

estoy usando un tipo de excepción genérica, aun no entiendo del todo como funciona pero fue lo que me han enseñado

3.- ¿Qué otras opciones consideraste y descartaste?

si consideré ignorar el error pero eso iba en contra de mi práctica así que trate de usar una excepción aunque como dije aun no la comprendo del todo así que agradecería que pudieras explicarme de vuelta cómo es que funcionan

4.- ¿En qué escenario tu decisión sería perfecta/mala?

mi decisión sería buena ahora que trabajo en un entorno pequeño y seguro, pero sería mala si trabajase con tipos de almacenamiento

decisión 5:

1.- ¿Por qué elegiste copiar elementos uno por uno?

elegí copiar elementos uno por uno ya que a trabajar con arreglos pequeños no me estoy preocupando tanto por el rendimiento con datos grandes

2.- ¿Qué otras estrategias consideraste (o podrías considerar)?

no pense en utilizar alguna otra estrategia, ya que copiar era la que mas se me facilitaba

3.- ¿En qué escenario tu decisión sería óptima?

definitivamente la el escenario de mi situacion mas optima es con arreglos pequeños

4.- ¿Cómo impactaría esta decisión en distintos contextos?

esto principalmente impactaria en la memoria y tambien en el tiempo en el que se ejecutan los datos

- Matriz de contexto vs. configuración ideal (al menos 4 contextos diferentes)

Contexto	Capacidad inicial	Factor de crecimiento	Reducción de capacidad	Manejo de indice fuera de rango	copia de elementos
Ejercicio práctico con arreglos pequeños (tu caso actual)	10	2x	No se debería de implementar	Excepción genérica	Uno por uno: simple y clara para pequeños tamaños
Sistema embebido con solo 512KB de RAM	1	1.5x	Sí, con límite mínimo	Valor especial (-1/null): Evita costos de excepción	Copia masiva: optimizado para reducir costos de redimensionar
Servidor con terabytes de RAM	64	3x o más	No se debería de implementar	Excepción específica: Manejar errores detallados	Copia masiva: Reducir latencias en operaciones grandes
Aplicación en tiempo real	Depende del caso	1.25x – 1.5x	Si pero de forma gradual	Priorizar estabilidad	Buffers paralelos: Reducir impacto en eventos críticos.

- Una decisión que cambiarías de tu implementación original y por qué

utilizar alguna otra técnica que no sea copiar los datos, cambiaría esa debido a que si bien con datos pequeños es útil pero con muchos más datos o más capacidad sería poco sofisticada

6.- Identificación de Patrones en Problemas de Arreglos

- "Tarjetas de referencia" para cada patrón con: nombre, señales de detección, idea general

Dos punteros

Señales:

Usas arreglos ordenados o puedes ordenarlos

Comparamas elementos desde extremos o índices específicos del arreglo

Indicaciones: "Encuentra un par...", "Empareja elementos...", "Relaciones entre índices"

Idea General:

Utiliza dos índices inicializados en posiciones estratégicas (extremos)

Ajusta los punteros dinámicamente según la condición (avanzar, retroceder o ambos)

Optimiza búsquedas y emparejamientos en tiempo $O(n)$

Ventana deslizante

Señales:

Trabajas con subarreglos consecutivos o intervalos dinámicos

Indicaciones: "Maximiza/minimiza en un rango", "Encuentra subarreglos que cumplan algo"

Idea General:

Mantén una "ventana" móvil (subarreglo) con extremos que se ajustan dinámicamente

Añade un elemento al entrar en la ventana y elimina uno al salir

Reutiliza resultados parciales, logrando tiempo $O(n)$

IN-PLACE MODIFICATION

Señales:

El problema pide modificar directamente el arreglo.

Indicaciones: "Hazlo sin memoria extra", "Modifica en su lugar", "Reorganiza elementos"

Idea General:

Usa índices para sobrescribir y procesar elementos directamente en el arreglo

Decide mover, eliminar o reorganizar mientras recorres, sin usar estructuras adicionales

Optimiza espacio con complejidad O(1)

PREFIJO/SUFIJO (Prefix/Suffix)

Señales:

Consultas repetitivas sobre subarreglos o rangos acumulativos

Indicaciones: "Calcula sumas/productos acumulados", "Evalúa entre rangos (i, j)"

Idea General:

Precalcula un arreglo acumulativo (prefijos/sufijos) con valores del inicio o final

Usa estos valores para responder consultas de rango en tiempo O(1).

Ideal para optimizar múltiple acceso a datos acumulados

- Para cada problema del Reto 4, indica qué patrón(es) usaste o podrías usar

Problema 1: Dos Punteros

Problema 2: Ventana Deslizante

Problema 3: Prefijo

Problema 4: Dos punteros

- Lista de 2-3 palabras clave que te harán pensar en cada patrón

Dos Punteros

- Ordenado
- Emparejar elementos

Ventana Deslizante

- Subarreglos consecutivos
- Máximo/Mínimo

- Tamaño fijo

In-Place Modification

- modificacion directa
- Reorganizar elementos

Prefijo/Sufijo

- Rango acumulado
- Producto en un intervalo
- Precalcular

Checkpoint Metacognitivo - Fase REFLEXIONA

- ¿Qué decisión de diseño me costó más justificar? ¿Qué dice eso sobre mi comprensión?

me costo mucho la de las excepciones, aunque mas que nada fue por que es algo que no habia visto hasta el momento

- ¿Qué patrón de los explorados era nuevo para mí? ¿Puedo ahora reconocerlo en un problema?

todos son nuevos para mi, y si puedo reconocerlos siempre y cuando tenga mis apuntes a un lado, necesito estudiar mas eso

- Si tuviera que enseñar "arreglos dinámicos" a un compañero, ¿por dónde empezaría?

explicando la teoría de como es que estos funcionan

¿Qué conexión inesperada descubrí entre los conceptos de esta semana?

como es que se ven aplicadas las distintas complejidades en los arreglos dinamicos