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 + (0×4) = 1000	[5] (arreglo 1)
1004	d base + (1×4) = 1004	[10] (arreglo 2)
1008	d base + (2×4) = 1008	[15] (arreglo 3)
1012	d base + (3×4) = 1012	[20] (arreglo 4)
1016	d base + (4×4) = 1016	[25] (arreglo 5)
1020	d base + (5×4) = 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[log2n]−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 lenta

  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 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 estructures 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--; // 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(índice, elemento)	O(n)	O(1)	Desplaza los elementos desde el índice especificado hacia la derecha y luego inserta el nuevo elemento allí.
eliminar(índice)	O(n)	O(n)	Desplaza los elementos hacia la izquierda para rellenar 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 ← k mod 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 ← arr[inicio]
    arr[inicio] ← arr[fin]
    arr[fin] ← temp

    inicio ← inicio + 1
    fin ← 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 ← 0

Para j ← 1 Hasta n-1 Hacer
   Si nums[j] =/ nums[i] Entonces
      i ← i + 1
      nums[i] ← 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 ← 0

   Para i ← 0 Hasta n-1 Hacer
      Si nums[i] ≠ 0 Entonces
         Intercambiar nums[lastNonZeroIndex] ↔ nums[i]
         lastNonZeroIndex ← lastNonZeroIndex + 1
      FinSi
   FinPara
FinAlgoritmo
```

Problema 4 encontrar elemento mayoritario:

```
   Entrada: arreglo nums de tamaño n
   Salida: elemento mayoritario

   Inicializar candidate ← Ninguno
   Inicializar count ← 0

Para cada num en nums Hacer
      Si count = 0 Entonces
         candidate ← 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 solucion 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 ahorre el hecho de usar un arreglo auxiliar, asi 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 teorica pero por alguna razon memorizar como se aplican de forma practica 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 sabia 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 pedian los problemas tenia que implementar cosas que pase muy por encima el semestre pasado, asi que si bien en esto sufri demasiado, lo visto en esta fase solo me motiva a estudiar mas respecto a la programacion de codigos

- ¿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 reduccion de capacidad por que como solo es un ejercicio practico y sencillo realmente no lo considere necesario, haciendo la implementacion mas 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 se que pase pero asumo que de alguna forma habria problemas de la gestion de memoria, pero como dije lo que estoy haciendo es muy simple asi 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 seria un problema solo depender de la latencia seria cuando tenemos un arreglo inicial pequeño como es mi caso, pero como dije estoy trabajando con arreglos pequeños asi 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 por que asi 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 considere ignorar el error pero eso iba en contra de mi practica asi que trate de usar una excepción aunque como dije aun no la comprendo del todo asi que agradeceria que pudieras explicarme de vuelta como es que funcionan

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

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

decisión 5:

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

elegi copiar elementos de 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 tecnica que no sea copiar los datos, cambiaria esa debido a que si bien con datos pequeños es util pero con muchos mas datos o mas capacidad seria poco sotisficada

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
Comparas 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

7.- Cazador de Bugs

- Para cada implementación: bug identificado, línea específica, corrección propuesta
  Bug 1:
  Definir nuevoArreglo <- crearArreglo(nuevaCapacidad)

  Solución 1:
  Definir nuevoArreglo <- creararreglo(nuevaCapacidad)

```
para i <- 0 hasta nuevacapacidad - 1 Hacer
   nuevoarreglo[i] <- nulo
fin para
```

Bug 2:

```
si indice < 0 o indice > tamano entonces
```

Solución 2:

```
Si indice < 0 o indice >= tamano + 1 entonces
   mostrar "Índice fuera de rango"
   finproceso
Fin Si
```

Bug 3:

```
si indice < 0 o indice >= tamano entonces
```

Solución 3:

```
si indice < 0 o indice > tamano - 1 entonces
   mostrar "Índice fuera de rango"
   finproceso
fin si
```

Bug 4:

```
definir estadoredimension <- verdadero
```

Solución 4:

```
subproceso iniciar(puederedimensionar <- verdadero)
   definir arreglo <- creararreglo(10)
   definir capacidad <- 10
   definir tamano <- 0
   estadoredimension <- puederedimensionar
fin subproceso
```

- Caso de prueba mínimo que revelaría cada bug

  Caso 1:

  ```
  para i <- 1 hasta 15 hacer
    agregar(i)
  finpara
  ```

  Caso 2:

  ```
  Para i <- 1 hasta 5 hacer
     agregar(i)
  finpara
  ```

```
insertar(tamano, 99)
```

Caso 3:

```
para i <- 1 hasta 3 hacer
   agregar(i)
finpara
eliminar(tamano - 1)
```

Caso 4:

```
Definir AD <- nuevo arreglodinamico()
AD.estadoredimension <- falso
para i <- 1 hasta 15 hacer
   AD.agregar(i)
fin para
```

- Ranking de los bugs de más fácil a más difícil de detectar, con justificación

Top 4: bug en eliminar(índice)
este se me hizo el mas facil de identificar debido a que con intentar eliminar el ultimo elemento de un indice nos marcara un error inmediatamente, por lo mismo es muy facil notar el bug y corregirlo

Top 3: bug en insertar(índice, elemento)
es similar al anterior, ya que al ingresar un elemento valido en el indice nos arrojaria error, lo que hace que sea facil de detectar el error y poder corregirlo

Top 2: bug en agregar() (redimensionamiento)
este es mas dificil de detectar que los otros dos debido a que no se puede detectar a menos que la capacidad inicial este llena, mas aparte los valores “basura” que genera son mas sutiles que los otros bugs, por lo que es mas dificil detectarlo

Top 1: bug en Inicialización
este es el mas dificil de detectar debido a que solo se manifiesta cuando se intenta desactivar el redimensionamiento