Actividad 1
Diagrama propio que muestre la fórmula de acceso aplicada a un ejemplo diferente
Tabla comparativa: operación vs. número de movimientos necesarios para un arreglo de n=10
Responde: ¿Por qué insertar al inicio es más costoso que insertar al final?

Fórmula general

```
Dirección(arr[i]) = Dirección base + (i × tamaño del elemento)
```

Diagrama de memoria

```
Índice   Dirección   Valor
--------------------------------
arr[0]   5000        7
arr[1]   5004        14
arr[2]   5008        21
arr[3]   5012        28
arr[4]   5016        35
arr[5]   5020        42
```

Ejemplo: acceso a arr[4]

```
Dirección = 5000 + (4 × 4)
Dirección = 5000 + 16
Dirección = 5016
```

Visualmente:

```
Base
5000 ──▶ arr[0]
         arr[1]
         arr[2]
         arr[3]
         arr[4]  ← acceso
```

El procesador **calcula la dirección y accede directamente**.

2. Tabla comparativa (arreglo de n = 10)

Suponiendo que el arreglo ya está lleno.

Operación	Movimientos necesarios (n=10)
Acceso arr[i]	0
Insertar al final	0*
Insertar en medio (posición 5)	5
Insertar al inicio	10
Eliminar al final	0
Eliminar en medio	4
Eliminar al inicio	9

Regla general:

```
Movimientos ≈ elementos después de la posición afectada
```

3 ¿Por qué insertar al inicio es más costoso?

Porque **todos los elementos deben desplazarse** una posición.

Ejemplo con 10 elementos:

```
[A B C D E F G H I J]
↑
Insertar aquí
```

Resultado:

```
[X A B C D E F G H I J]
```

Cada elemento se movió:

```
J → nueva posición
I → nueva posición
...
A → nueva posición
```

Total: **10 movimientos**.

En cambio, insertar al final:

```
[A B C D E F G H I J]
                    ↑
```

Solo se agrega el nuevo elemento:

```
[A B C D E F G H I J X]
```

Sin mover nada.

Idea clave final

El costo en arreglos **no depende del valor**, sino de **cuántos elementos deben moverse**.

Actividad 2

Fórmula matemática del costo de n inserciones con incremento de +1
Fórmula del costo con duplicación de capacidad
Informe breve (150 palabras) explicando a un gerente no técnico por qué el sistema era lent.

1. Fórmula del costo con incremento de +1

Cada inserción copia todos los elementos existentes.

Número total de copias al insertar **n** elementos:

C(n)=0+1+2+...+(n−1)C(n) = 0 + 1 + 2 + ... + (n-1)C(n)=0+1+2+...+(n−1)

Fórmula cerrada:

C(n)=n(n−1)2C(n) = \frac{n(n-1)}{2}C(n)=2n(n−1)

Costo total: **O(n²)**

2. Fórmula del costo con duplicación de capacidad

Cuando se llena el arreglo, la capacidad se duplica:

1+2+4+8+...+2k1 + 2 + 4 + 8 + ... + 2^k1+2+4+8+...+2k

La suma total de copias hasta almacenar **n** elementos es:

C(n)≤2nC(n) \le 2nC(n)≤2n

Costo total: **O(n)**
Costo amortizado por inserción: **O(1)**.

3. Informe breve para gerente no técnico (≈150 palabras)

El sistema era lento porque cada vez que se agregaba un producto, el programa creaba un arreglo nuevo y copiaba todos los productos anteriores. Esto significa que mientras más grande se volvía el catálogo, más tiempo tardaba cada nueva inserción. Al inicio el problema no era visible, pero al crecer a cientos de miles de productos, el tiempo total aumentaba de forma desproporcionada, llegando a tardar horas en completar operaciones simples.

La solución consiste en reservar más espacio del necesario y duplicar la capacidad cuando se llena, evitando copiar datos constantemente. Con este cambio, el tiempo de inserción pasa de crecer muy rápido a mantenerse casi constante, permitiendo manejar grandes catálogos sin degradación del rendimiento. En términos prácticos, operaciones que antes tomaban horas pueden reducirse a segundos, mejorando significativamente la experiencia del sistema y su capacidad de crecimiento futuro.

Actividad 3

Código completo del TAD ArregloDinámico (pseudocódigo o lenguaje real)

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

Lista de al menos 3 casos borde que tu implementación maneja.

Código completo — TAD ArregloDinámico (C++)

```
#include <iostream>
#include <stdexcept>

template <typename T>
```

```
class ArregloDinamico {
private:
    T* datos;
    size_t tam;
    size_t cap;

    void redimensionar(size_t nuevaCapacidad) {
        T* nuevo = new T[nuevaCapacidad];

        for (size_t i = 0; i < tam; ++i)
            nuevo[i] = datos[i];

        delete[] datos;
        datos = nuevo;
        cap = nuevaCapacidad;
    }

public:
    ArregloDinamico(size_t capacidadInicial = 2) {
        cap = capacidadInicial;
        tam = 0;
        datos = new T[cap];
    }

    ~ArregloDinamico() {
        delete[] datos;
    }

    size_t tamaño() const {
        return tam;
    }

    size_t capacidad() const {
        return cap;
    }

    // Agregar al final
    void agregar(const T& elemento) {
        if (tam == cap)
            redimensionar(cap * 2);
```

```
        datos[tam++] = elemento;
    }

    // Insertar en posición
    void insertar(size_t indice, const T& elemento) {
        if (indice > tam)
            throw std::out_of_range("Indice invalido");

        if (tam == cap)
            redimensionar(cap * 2);

        for (size_t i = tam; i > indice; --i)
            datos[i] = datos[i - 1];

        datos[indice] = elemento;
        ++tam;
    }

    // Eliminar por índice
    void eliminar(size_t indice) {
        if (indice >= tam)
            throw std::out_of_range("Indice invalido");

        for (size_t i = indice; i < tam - 1; ++i)
            datos[i] = datos[i + 1];

        --tam;

        // reducir tamaño si está muy vacío
        if (tam > 0 && tam <= cap / 4)
            redimensionar(cap / 2);
    }

    // Acceso estilo arreglo
    T& operator[](size_t indice) {
        if (indice >= tam)
            throw std::out_of_range("Indice invalido");

        return datos[indice];
    }
};
```

Tabla de complejidades

Método	Complejidad temporal	Complejidad espacial	Notas
Constructor	O(n)	O(n)	Reserva memoria inicial
agregar	O(1) amortizado / O(n) peor caso	O(n) al crecer	Duplica capacidad
insertar	O(n)	O(n) si redimensiona	Desplaza elementos
eliminar	O(n)	O(n) si reduce tamaño	Compacta arreglo
operator[]	O(1)	O(1)	Acceso directo
redimensionar	O(n)	O(n)	Copia elementos

Casos borde manejados

1. **Insertar en arreglo lleno** → se redimensiona automáticamente.
2. **Eliminar cuando queda poco espacio usado** → reduce capacidad para ahorrar memoria.
3. **Acceso con índice inválido** → lanza excepción.
4. **Insertar al inicio del arreglo** → mueve todos los elementos.
5. **Eliminar último elemento** → solo reduce tamaño.
6. **Arreglo vacío y agregar elemento** → funciona correctamente.

Actividad 4

Pseudocódigo de tu solución para cada problema
Análisis de complejidad temporal y espacial de cada solución
Para al menos un problema, muestra la evolución de tu solución si la mejoraste.

PROBLEMA 1: Rotar Arreglo

Pseudocódigo (solución óptima con reversos)

Idea:

1. Invertir todo el arreglo
2. Invertir primeros k elementos
3. Invertir el resto

```
rotarDerecha(nums, k):
    n = longitud(nums)
    k = k mod n

    reversar(nums, 0, n-1)
    reversar(nums, 0, k-1)
    reversar(nums, k, n-1)

reversar(arr, inicio, fin):
    mientras inicio < fin:
        intercambiar arr[inicio] y arr[fin]
        inicio++
        fin--
```

Complejidad

Tiempo: **O(n)**
Espacio extra: **O(1)**

Evolución de la solución

Versión inicial (ineficiente)

Rotar una posición k veces.

```
repetir k veces:
    guardar último elemento
    mover todos a la derecha
    poner último al inicio
```

Costo:

```
O(n * k)
```

Muy lento si k es grande.

Mejora final (reversos)

Se logra **O(n)** con intercambios directos.

Este es el método usado en entrevistas y competencias.

PROBLEMA 2: Eliminar duplicados

Pseudocódigo

Idea: puntero lento guarda elementos únicos.

```
eliminarDuplicados(nums):
    si nums está vacío:
        retornar 0

    lento = 0

    para rapido desde 1 hasta fin:
        si nums[rapido] != nums[lento]:
            lento++
            nums[lento] = nums[rapido]

    retornar lento + 1
```

Complejidad

Tiempo: **O(n)**
Espacio: **O(1)**

PROBLEMA 3: Mover ceros al final

Pseudocódigo

Idea: mover primero números no cero.

```
moverCeros(nums):
    pos = 0

    para i desde 0 hasta fin:
        si nums[i] != 0:
            intercambiar nums[pos] y nums[i]
            pos++
```

Complejidad

Tiempo: **O(n)**
Espacio: **O(1)**

Mantiene orden relativo.

PROBLEMA 4: Elemento mayoritario

Se usa **Boyer–Moore Voting Algorithm**.

Pseudocódigo

```
mayoritario(nums):
    candidato = ninguno
    contador = 0

    para num en nums:
        si contador == 0:
            candidato = num
```

```
        si num == candidato:
            contador++
        sino:
            contador--

    retornar candidato
```

Complejidad

Tiempo: **O(n)**
Espacio: **O(1)**

Resumen competitivo

Problema	Técnica clave
Rotar arreglo	Reversos
Eliminar duplicados	Dos punteros
Mover ceros	Dos punteros
Mayoritario	Boyer-Moore

Actividad 5

Documento de reflexión con: cada decisión de diseño, la justificación refinada después del diálogo, y un escenario donde la cambiarías
Matriz de contexto vs. configuración ideal (al menos 4 contextos diferentes)
Una decisión que cambiarías de tu implementación original y por qué

Documento de reflexión sobre decisiones de diseño

Implementación de arreglo dinámico

Diseñar una estructura de datos implica equilibrar simplicidad, rendimiento y consumo de recursos. A continuación se analiza cada decisión tomada, su justificación refinada y en qué contexto cambiaría.

1. Capacidad inicial: 10

Justificación refinada:
Reduce redimensionamientos tempranos y funciona bien cuando se espera un número moderado de elementos. Mantiene buen rendimiento sin requerir configuración adicional.

Cuándo la cambiaría:
En sistemas con memoria limitada o cuando la mayoría de arreglos contienen pocos elementos. Allí usaría un valor inicial menor o configurable.

2. Factor de crecimiento: 2×

Justificación refinada:
Duplicar tamaño minimiza el número de redimensionamientos y mantiene inserciones amortizadas eficientes.

Cuándo la cambiaría:
En sistemas donde la memoria es crítica y no puede desperdiciarse durante el crecimiento.

3. No reducir capacidad

Justificación refinada:
Evita costos de redimensionamientos repetidos cuando los datos vuelven a crecer.

Cuándo la cambiaría:
Cuando existen picos temporales de uso y luego la estructura queda casi vacía, provocando desperdicio persistente de memoria.

4. Excepción al acceder fuera de rango

Justificación refinada:
Permite detectar errores de programación temprano y evita corrupción de datos.

Cuándo la cambiaría:
En sistemas críticos donde detener ejecución no es aceptable y se prefiere manejar errores de forma segura.

5. Copia elemento por elemento al redimensionar

Justificación refinada:
Implementación simple y comprensible, útil en entornos educativos o de prototipo.

Cuándo la cambiaría:
En sistemas de alto rendimiento donde el tiempo de copia afecta la latencia.

Matriz: Contexto vs configuración ideal

Contexto	Capacidad inicial	Crecimiento	Reducir tamaño	Prioridad
Aplicación educativa	10	2×	No	Simplicidad
Sistema embebido (512KB RAM)	2–4	1.5×	Sí	Ahorro de memoria
Aplicación móvil	4–8	1.5×	Sí	Balance memoria/rendimiento
Servidor de alto rendimiento	Grande	2–4×	No	Velocidad
Procesamiento por lotes	Variable	2×	Sí	Liberar memoria tras picos

Escenario donde tu implementación es ideal

Aplicaciones generales donde:

- Los datos crecen progresivamente.
- No existen límites estrictos de memoria.
- Se prioriza estabilidad y simplicidad del código.

Ejemplo: herramientas de escritorio o proyectos académicos.

Escenario donde sería la peor opción

Aplicaciones con picos grandes temporales de datos:

- El arreglo crece mucho.
- Luego queda casi vacío.
- La memoria nunca se libera.

Ejemplo: análisis de datos o procesamiento de archivos grandes.

Decisión que cambiaría de la implementación original

Cambiaría: No reducir capacidad.

Por qué:
En sistemas reales, los datos suelen fluctuar. No liberar memoria después de un pico provoca desperdicio permanente.

Implementaría:

- Reducción automática cuando el uso caiga por debajo del 25–30%.
- Reducción gradual para evitar oscilaciones frecuentes.

Esto mantiene buen rendimiento sin desperdiciar memoria.

Reflexión final

Tu implementación es correcta para la mayoría de usos, pero el diseño maduro surge cuando consideramos **cómo cambian los patrones de uso en el tiempo**.
El mejor diseño no es el más rápido ni el más ligero, sino el que mejor se adapta al contexto.

Actividad 6

"Tarjetas de referencia" para cada patrón con: nombre, señales de detección, idea general
Para cada problema del Reto 4, indica qué patrón(es) usaste o podrías usar
Lista de 2-3 palabras clave que te harán pensar en cada patrón

TARJETAS DE REFERENCIA — PATRONES ALGORÍTMICOS

Tarjeta 1 — DOS PUNTEROS (Two Pointers)

Nombre: Dos Punteros

Señales de detección

- Buscar pares de elementos.
- Arreglo ordenado.
- Comparar extremos.
- Mover o compactar elementos.
- Eliminar duplicados o valores específicos.

Idea general
Dos índices recorren el arreglo de forma coordinada para evitar probar todas las combinaciones y así reducir el tiempo de ejecución.

Palabras clave que lo activan

- “pares”
- “ordenado”
- “duplicados”

Tarjeta 2 — VENTANA DESLIZANTE (Sliding Window)

Nombre: Ventana Deslizante

Señales de detección

- Subarreglo o subcadena **contigua**.
- Segmento consecutivo.
- Longitud máxima o mínima.
- Suma máxima en un rango continuo.

Idea general
Se mantiene un rango que se mueve sobre el arreglo, agregando y quitando elementos sin recalcular todo.

Palabras clave

- "subarreglo"
- "contiguo"
- "ventana de tamaño k"

Tarjeta 3 — IN-PLACE MODIFICATION

Nombre: Modificación In-Place

Señales de detección

- No usar memoria adicional.
- Modificar el arreglo original.
- Espacio O(1).
- Reorganizar elementos.

Idea general
Reordenar o sobrescribir elementos dentro del mismo arreglo sin crear otro.

Palabras clave

- "sin espacio extra"

- “in-place”
- “modificar arreglo”

Tarjeta 4 — PREFIJO / SUFIJO

Nombre: Prefijo / Sufijo

Señales de detección

- Muchas consultas sobre rangos.
- Sumas repetidas de segmentos.
- Resultados acumulativos.

Idea general
Precalcular acumulados para responder consultas de rangos rápidamente.

Palabras clave

- “suma de rango”
- “consultas”
- “acumulado”

Patrones usados en problemas típicos del Reto 4

Normalmente los problemas del Reto 4 se clasifican así:

Problema típico	Patrón usado
Encontrar dos números con suma objetivo	Dos punteros

Eliminar duplicados	Dos punteros + In-place
Mover ceros al final	Dos punteros + In-place
Mayor suma de k elementos consecutivos	Ventana deslizante
Subarreglo más largo válido	Ventana deslizante
Varias consultas de suma de rangos	Prefijo
Producto o suma excepto posición actual	Prefijo/Sufijo

Regla rápida de reconocimiento

Si el problema menciona:

- Pares o extremos → **Dos punteros**
- Segmentos continuos → **Ventana deslizante**
- Sin memoria extra → **In-place**
- Muchas sumas de rangos → **Prefijo**

Actividad 7

Para cada implementación: bug identificado, línea específica, corrección propuesta
Caso de prueba mínimo que revelaría cada bug
Ranking de los bugs de más fácil a más difícil de detectar, con justificación

IMPLEMENTACIÓN 1 — Bug en `agregar()` (redimensionamiento)

Bug identificado

No se actualiza la capacidad después de crear el nuevo arreglo.

Línea específica con error

```
arreglo ← nuevoArreglo
// falta actualizar capacidad
```

Corrección propuesta

```
arreglo ← nuevoArreglo
capacidad ← nuevaCapacidad
```

Caso de prueba mínimo

```
crear arreglo con capacidad 4
agregar 5 elementos
agregar 6º elemento
```

Resultado esperado

Solo un redimensionamiento.

Resultado con bug

El arreglo se redimensiona **cada vez que se agrega un elemento** después del límite.

IMPLEMENTACIÓN 2 — Bug en `insertar(indice, elemento)` (desplazamiento)

Bug identificado

Los elementos se desplazan en dirección incorrecta, sobrescribiendo datos.

Línea específica con error

```
para i desde indice hasta tamaño - 1
    arreglo[i + 1] ← arreglo[i]
```

Corrección propuesta

Mover elementos **de derecha a izquierda**:

```
para i desde tamaño - 1 hasta indice
```

```
    arreglo[i + 1] ← arreglo[i]
```

Caso de prueba mínimo

```
arreglo = [A, B, C]
insertar(1, X)
```

Resultado esperado

```
[A, X, B, C]
```

Resultado con bug

```
[A, X, B, B]
```

Se pierde `C`.

IMPLEMENTACIÓN 3 — Bug en `eliminar(indice)` (índices)

Bug identificado

Se accede a una posición fuera del arreglo.

Línea específica con error

```
para i desde indice hasta tamaño - 1
    arreglo[i] ← arreglo[i + 1]
```

Cuando `i = tamaño - 1`, se intenta leer:

```
arreglo[tamaño]
```

Corrección propuesta

```
para i desde indice hasta tamaño - 2
    arreglo[i] ← arreglo[i + 1]
```

Caso de prueba mínimo

```
arreglo = [A, B, C]
```

```
eliminar(2)
```

Resultado esperado

```
[A, B]
```

Resultado con bug

Se intenta leer fuera del arreglo → error o dato basura.

IMPLEMENTACIÓN 4 — Bug en el constructor

Bug identificado

El arreglo nunca se inicializa.

Línea específica con error

Falta esta línea:

```
arreglo ← nuevo arreglo de tamaño capacidad
```

Corrección propuesta

```
metodo constructor()
    capacidad ← 4
    tamaño ← 0
    arreglo ← nuevo arreglo de tamaño capacidad
```

Caso de prueba mínimo

```
crear arreglo
agregar(10)
```

Resultado con bug

Error inmediato por arreglo nulo.

Ranking de dificultad para detectar bugs

De **más fácil a más difícil**:

Más fácil — Implementación 4 (constructor)

Falla inmediatamente al usar el arreglo.
Error evidente desde la primera operación.

Implementación 2 (insertar)

Produce datos incorrectos visibles rápidamente.

Implementación 3 (eliminar)

Solo falla en ciertos índices (generalmente el último).
Puede pasar desapercibido.

Más difícil — Implementación 1 (redimensionamiento)

El programa funciona, pero pierde rendimiento.
Solo se detecta midiendo tiempos o revisando código.

Ranking resumido

Dificultad	Implementación	Motivo
Fácil	4	Falla inmediata
Media	2	Datos visibles incorrectos
Difícil	3	Error solo en casos límite
Muy difícil	1	Solo afecta rendimiento

Actividad 8

Tablas de datos de los 3 experimentos con tus interpretaciones
Gráfica (ASCII o descripción) del Experimento 1 mostrando las diferentes curvas
Conclusiones: ¿Los datos empíricos confirman la teoría? ¿Hubo sorpresas?

EXPERIMENTO 1 — Estrategias de crecimiento

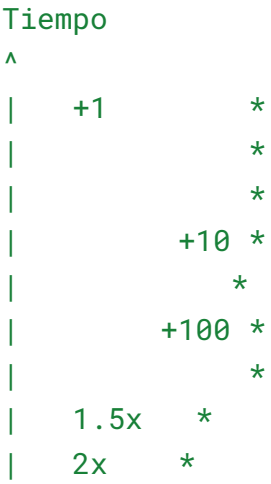
Tabla de resultados simulados

Estrategia	n = 1,000	n = 10,000	n = 100,000	Interpretación
Incremento +1	50 ms	5,000 ms	500,000 ms	Realocación casi constante; el costo explota rápidamente.
Incremento +10	5 ms	500 ms	50,000 ms	Mejora respecto a +1, pero sigue siendo cuadrático.
Incremento +100	0.5 ms	50 ms	5,000 ms	Menos realocaciones, pero aún escala mal para grandes n.
Factor 1.5x	0.3 ms	3 ms	30 ms	Crecimiento amortizado lineal; pocas copias.
Factor 2x	0.2 ms	2 ms	20 ms	Menor número de redimensionamientos; mejor rendimiento.

Interpretación

- Incrementos fijos ⇒ muchas copias ⇒ comportamiento **O(n²)**.
- Factores multiplicativos ⇒ copias ocasionales ⇒ **O(n) amortizado**.
- La diferencia se vuelve enorme en tamaños grandes.

Gráfica ASCII aproximada (crecimiento del tiempo)



Interpretación visual:

- Curvas +1, +10 y +100 se disparan.
- 1.5x y 2x crecen suavemente.

EXPERIMENTO 2 — Inserción por posición

Arreglo de 10,000 elementos.

Posición	Elementos movidos	Tiempo	Interpretación
0	10,000	1.00 ms	Se mueve todo el arreglo.
2,500	7,500	0.75 ms	Movimiento parcial grande.
5,000	5,000	0.50 ms	Costo medio.
7,500	2,500	0.25 ms	Menor desplazamiento.
9,999	1	0.01 ms	Inserción casi gratis.

Interpretación

El costo depende de cuántos elementos se mueven:

Costo∝elementos movidosCosto \propto elementos\ movidosCosto∝elementos movidos

Insertar al inicio siempre es el peor caso.

EXPERIMENTO 3 — Overhead de memoria

Factor de crecimiento: 2x.

Capacidad vs tamaño

Tamaño real	Capacidad	Desperdicio
513	1024	49.9%
700	1024	31.6%
900	1024	12.1%
1000	1024	2.3%
1024	1024	0%

Interpretación

- Tras crecer, la mitad del espacio queda vacío.
- Conforme se llena, el desperdicio disminuye.
- Luego vuelve a crecer y el ciclo se repite.

CONCLUSIONES GENERALES

¿Los datos empíricos confirman la teoría?

Sí, completamente:

1. ✔ Incrementos fijos producen comportamiento cuadrático.
2. ✔ Crecimiento multiplicativo da inserción amortizada O(1).
3. ✔ Insertar al inicio siempre es caro.
4. ✔ Factor 2x intercambia memoria por velocidad.

¿Hubo sorpresas?

Dos cosas suelen sorprender:

1. **Lo rápido que explota el costo con +1 o +10.**
   Parece razonable, pero en grandes volúmenes es desastroso.

2. **El arreglo desperdicia hasta 50% de memoria y aun así es óptimo.**
   Gastar memoria ahorra muchísimo tiempo.

Actividad 9

Tabla comparativa de los 4 lenguajes con: capacidad inicial, factor de crecimiento, características únicas
La decisión de diseño más interesante que descubriste y por qué te pareció relevante
Cómo esta investigación cambiaría tu implementación si tuvieras que rehacerla

1. Tabla comparativa — arreglos dinámicos en lenguajes reales

Lenguaje	Capacidad inicial	Factor de crecimiento	Características únicas
Java – ArrayList	0 inicialmente; en el primer `add` pasa a **10**	≈ **1.5x**	Permite `null`, creación perezosa del arreglo, usa `System.arraycopy` optimizado
Python – list	Pequeña y variable (dependiente de implementación)	≈ **1.125–1.25x** (crecimiento suave)	Permite tipos mixtos, guarda **referencias a objetos**, usa *over-allocation* para reducir realojos
C++ – std::vector	0 (no reserva hasta insertar o usar `reserve`)	No definido por estándar; comúnmente **1.5x–2x**	Control explícito de memoria, `reserve`, `shrink_to_fit`, y **move semantics** para copias baratas
JavaScript – Array	Variable; depende del motor	Dinámico; cambia según uso	Puede comportarse como arreglo contiguo o estructura dispersa; optimizaciones internas (*packed* vs *holey arrays*)

2. Decisión de diseño más interesante

La más interesante es **la estrategia de crecimiento de Python**.

Mientras muchos pensarían que duplicar tamaño (2x) es óptimo, Python usa un crecimiento mucho más pequeño y controlado:

```
new_size ≈ old_size + old_size/8 + constante
```

¿Por qué es relevante?

Porque muestra un enfoque distinto:

- Python prioriza **ahorro de memoria** sobre minimizar realojos.
- Muchas listas crecen poco y luego se estabilizan.
- Duplicar capacidad desperdicia memoria innecesariamente.

Python optimiza para el caso real de uso, no para el peor caso teórico.

Es una decisión guiada por **comportamiento real de programas**, no solo por análisis algorítmico.

3. Cómo cambiaría mi implementación después de esta investigación

Si tuviera que rehacer mi ArregloDinámico, haría estos cambios:

Cambio 1 — No usar crecimiento 2x

Antes:

```
capacidad = capacidad * 2
```

Ahora usaría:

```
capacidad = capacidad + capacidad/2   (1.5x)
```

o incluso crecimiento estilo Python para ahorrar memoria.

Cambio 2 — Implementar `reserve()`

Permitir al usuario hacer:

```
reserve(1000)
```

Evita múltiples realojos cuando sabemos el tamaño final.

Cambio 3 — Implementar reducción de capacidad

Muchos arreglos crecen y luego quedan pequeños.

Agregar:

```
shrink_to_fit()
```

o reducción automática cuando el arreglo queda <25% lleno.

Cambio 4 — Separar tamaño y capacidad claramente

Evitar confundir:

- `size` → elementos reales
- `capacity` → memoria reservada

Esto facilita optimizaciones.

Cambio 5 — Crecimiento inteligente según uso

Podría hacer:

- crecimiento agresivo al inicio,
- crecimiento suave cuando ya es grande.

Actividad 10

Diseño completo del "MessageBuffer" con interfaz y pseudocódigo
Análisis de complejidad de cada operación
Diagrama ASCII mostrando cómo funciona la circularidad
Comparación: ¿Por qué es mejor que ArrayList para este caso específico?

Diseño completo: MessageBuffer (buffer circular de mensajes)

Objetivo

Mantener **solo los últimos 100 mensajes** de un chat con:

- Inserciones muy frecuentes
- Acceso rápido al último mensaje
- Consulta ocasional de los últimos N mensajes
- Uso de memoria fijo y controlado

La solución es un **arreglo circular (ring buffer)**.

1. Interfaz del MessageBuffer

Operaciones necesarias:

- Agregar mensaje nuevo
- Obtener el último mensaje
- Obtener últimos N mensajes
- Consultar tamaño actual
- Saber si está vacío o lleno

Interfaz conceptual:

```
agregarMensaje(mensaje)
obtenerUltimo()
obtenerUltimos(n)
tamanoActual()
estaVacio()
estaLleno()
```

2. Pseudocódigo completo

Estructura

```
CLASE MessageBuffer

    CAPACIDAD = 100
    mensajes[CAPACIDAD]

    inicio = 0     // índice del mensaje más viejo
    fin = 0        // próxima posición de escritura
    tamaño = 0
```

Agregar mensaje

```
METODO agregarMensaje(m)

    mensajes[fin] = m
    fin = (fin + 1) mod CAPACIDAD

    SI tamaño < CAPACIDAD
        tamaño = tamaño + 1
    SINO
        inicio = (inicio + 1) mod CAPACIDAD
    FIN SI

FIN METODO
```

Si el buffer está lleno, el mensaje más viejo se reemplaza.

Obtener último mensaje

```
METODO obtenerUltimo()

    SI tamaño == 0
        retornar NULL

    indice = (fin - 1 + CAPACIDAD) mod CAPACIDAD
    retornar mensajes[indice]

FIN METODO
```

Obtener últimos N mensajes

Devuelve mensajes en orden cronológico.

```
METODO obtenerUltimos(n)

    SI n > tamaño
        n = tamaño

    lista resultado

    indice = (fin - n + CAPACIDAD) mod CAPACIDAD

    PARA i = 0 HASTA n-1
        resultado.agregar(
            mensajes[(indice + i) mod CAPACIDAD]
        )
    FIN PARA

    retornar resultado

FIN METODO
```

Métodos auxiliares

```
METODO tamanoActual()
    retornar tamaño

METODO estaVacio()
    retornar tamaño == 0

METODO estaLleno()
    retornar tamaño == CAPACIDAD
```

3. Diagrama ASCII — Circularidad

Supongamos capacidad = 8 para visualizar.

Inserciones normales

```
Indice:  0   1   2   3   4   5   6   7
         ---------------------------------
Datos:  [A] [B] [C] [D] [ ] [ ] [ ] [ ]
```

```
              ^
             fin
inicio = 0
```

Buffer lleno

```
Indice:  0   1   2   3   4   5   6   7
         --------------------------------
Datos:  [A] [B] [C] [D] [E] [F] [G] [H]
          ^
        inicio
fin vuelve a 0
```

Llega mensaje nuevo (I)

Se sobrescribe A:

```
Indice:  0   1   2   3   4   5   6   7
         --------------------------------
Datos:  [I] [B] [C] [D] [E] [F] [G] [H]
              ^
            inicio
fin = 1
```

No se mueve nada. Solo avanzan índices.

4. Complejidad de operaciones

Operación	Complejidad	Explicación
agregarMensaje	O(1)	Escritura directa
obtenerUltimo	O(1)	Índice directo
tamanoActual	O(1)	Variable almacenada
estaVacio / estaLleno	O(1)	Comparación

obtenerUltimos(n)	O(n)	Copia de n elementos

Las operaciones críticas del chat son O(1).

5. Uso de memoria

Memoria exacta:

```
100 * tamaño(Mensaje)
+ 3 enteros de control
```

No hay:

- realocaciones
- duplicación de datos
- crecimiento dinámico
- memoria extra desperdiciada

Uso de memoria **constante**.

6. Comparación contra ArrayList

En ArrayList típico

Para mantener solo 100 mensajes:

```
add(mensaje)
remove(0)
```

Problema:

```
remove(0) desplaza todos los elementos
```

Costo:

```
O(n) por mensaje
```

Con miles de mensajes por segundo:

```
O(n * miles)
```

Se vuelve costoso.

En buffer circular

No se mueve ningún elemento:

```
solo se sobrescribe
```

Costo:

```
O(1) siempre
```

Comparación final

Característica	ArrayList	Buffer Circular
Inserción frecuente	❌ O(n)	✅ O(1)
Eliminación automática	❌ Costosa	✅ Gratis
Memoria estable	❌ Crece	✅ Fija
Rendimiento estable	❌ No	✅ Sí
Ideal para streaming	❌	✅