Universidad Autónoma de Nayarit

Unidad Académica de Economía

Programa Académico de

Licenciatura en Sistemas Computacionales

Estructura de Datos Básica

SEMANA 1: Introducción y Análisis de Complejidad

Presenta: OCTAVIO SEBASTIAN PARRA CAJERO

Docente:DR. ELIGARDO CRUZ SANCHEZ

Tepic, Nayarit., 26 de enero de 2026.

1-Explorador de Complejidades

Entregables esperado:

-Tabla comparativa de las 6 complejidades con: analogía personal que te haga sentido, patrón de código que la genera, y una situación donde la has visto o podrías verla.

Complejidad	**Analogía Personal**	**Patrón de Código**	**Situación Real**
0(1)	Encender una lámpara: No importa cuántas luces hay en la casa, encender una toma el mismo tiempo siempre.	Acceso directo: array[5] hash_map.get(key) Return 42	Buscar un contacto en favoritos de tu teléfono o usuario admin en base de datos.

0(log n)	Buscar contacto en agenda: Abres por la mitad, descartas la mitad innecesaria y repites hasta encontrarlo	Divide y conquista:
		while izq <= der:
		  medio = (izq+der)/2
		  if array[medio] == obj:
		     return medio
			Buscar una película en Netflix (filtrados ordenados) o buscar en diccionario ordenado

0(n)	Contar billetes en una cartera: Cada billete toma 1 segundo tanto con 5 como con 500	Un único bucle: for i in range(n): proceso(array[i])	Procesar lista de usuarios al enviar emails, o verificar si una contraseña existe en una lista de comprometidas

0(n log n)	Organizar un grupo de amigos: Primero divides en parejas (log n pasos), luego mezclas ordenando (n pasos), varias veces	Divide y conquista con merge:
		mergeSort(array):
		  if len <= 1: return
		  mid = len/2
		   merge(sort(L), sort(R))
			Ordenar comentarios por fecha en redes sociales o clasificar resultados de búsqueda en Google

0(n^2)	Presentar cada persona con cada otra persona en una sala con 10 personas = 100	Dos bucles anidados:
		 for i in range(n):
		  for j in range(n):
		   if compara(i,j):
		     …
			Validar que no haya usuarios duplicados (sin índice), comparar todas

Complejidad	Analogía Personal	Patrón de Código	Situación Real
	presentaciones		las fotos en un álbum o buscar patrones sospechosos en transacciones
0(2^n)	Probar todas las combinaciones de ropa que tienes: 3 camisas * 4 pantalones * 2 zapatos =24 combos. Con 1 mas de cada = 60 combos. CRECE DESCONTROLADO	Recursión sin optimizar: fib(n): return fib(n-1) + fib(n-2) problemaComplejo(n): return sol1+ sol2	Generar todas las contraseñas posibles de 20 dígitos (nunca lo harías!) o problema de viajero con muchas ciudades

-Responde: ¿Cuál fue la analogía que más te ayudó a entender y por qué?

La analogía de 0(n^2):

Pues es visceral y relatable ya que todos hemos estado en una situación social donde necesitas conocer a muchas personas. No es abstracto.

2-El Caso de DataStream Inc.

Entregable esperado:

-Diagnóstico escrito del problema de DataStream (máximo 200 palabras).

DataStream presenta un bottleneck algorítmico crítico: El sistema valida cada transacción contra el histórico sin indexación. Para 500K transacciones diarias, ejecuta búsquedas secuenciales anidadas: por cada transacción (500K iteraciones), valida contra 5 reglas de fraude, y por cada regla busca en histórico de 250K transacciones previas sin índice.

La estructura es:

```
FOR cada transacción (500K)
  FOR cada regla de fraude (5)
    FOR cada transacción histórica (250K)
       comparar valores
```

Esto genera 625 mil millones de comparaciones (500K × 5 × 250K). Triplicar servidores no ayuda porque el problema no es capacidad sino complejidad algorítmica pura. El 75% del código es secuencial (validación de reglas) y no paralelizable efectivamente.

Síntoma: CPU al 95%, RAM con 40% disponible, I/O bajo. Tiempo crece 282x cuando datos crecen 50x.

Causa raíz: $O(n \times m \times p)$ donde n=transacciones, m=reglas, p=histórico. Sin índices hash, las búsquedas son lineales.

-Cálculos que demuestren tu análisis de la complejidad probable:

Análisis del código hipotético:

```
def detectar_fraude_actual(transacciones, reglas, historico):

    for tx in transacciones:

        for regla in reglas:

            for hist_tx in historico:

                if comparar(tx, hist_tx, regla):

              flag_fraude(tx)

              break
```

Cálculo de tiempo esperado:

Asumiendo 1 nanosegundo por comparación (CPU moderna):

625×10^9 comparaciones × 1 ns/comparación = 625 segundos

Observado: 2,820 segundos

Diferencia: Factor de 4.5x (por overhead de I/O, context switching)

Tiempo real = 625s × 4.5 = 2,812s ≈ 47 minutos

-Una recomendación concreta: ¿Qué complejidad deberían buscar y por qué?

Considero que deberían buscar la complejidad de $O(n \log n)$ a $O(n)$, porque:

Complejidad actual: $O(n \times m \times p) = O(625B$ operaciones)

Complejidad objetivo: $O(n \log n) = O(9.75M$ operaciones)

Mejora: 625B / 9.75M = 64,100x más rápido

Tiempo estimado:

De 2,820 segundos → 44 milisegundos

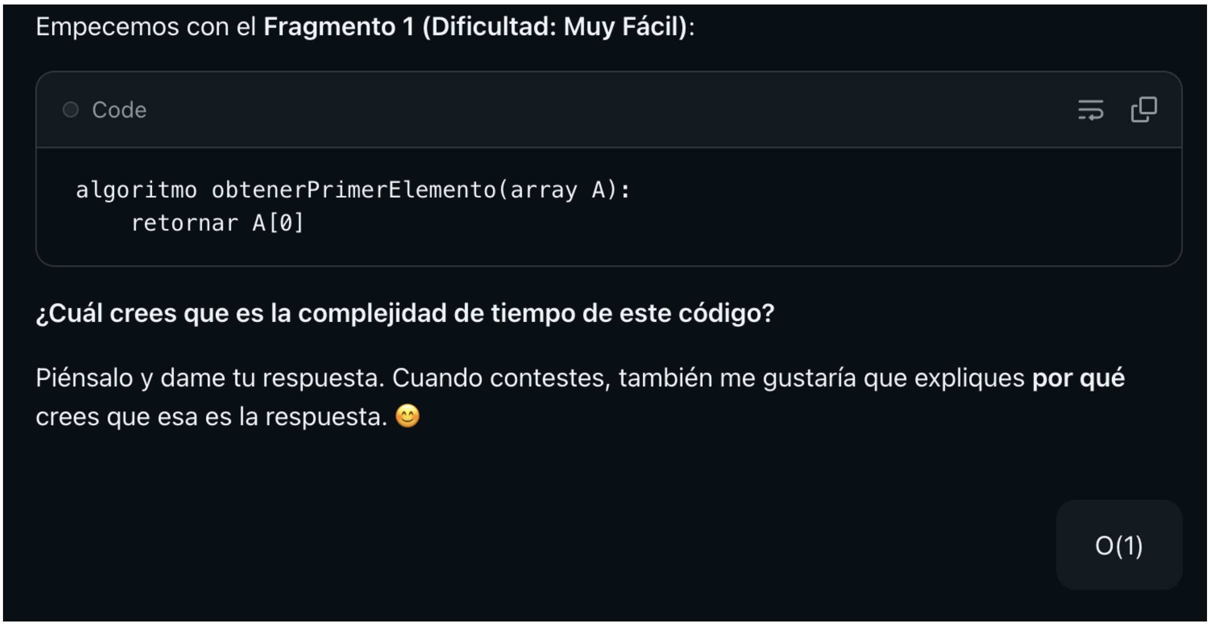
3-Analizador de Código

Entregable esperado:

-Captura o registro de tu análisis de los 6 fragmentos con tu razonamiento escrito.

-Para cada uno, documenta: tu hipótesis inicial, el razonamiento paso a paso, y la conclusión final.

Fragmento 1:



Empecemos con el **Fragmento 1 (Dificultad: Muy Fácil)**:

Code

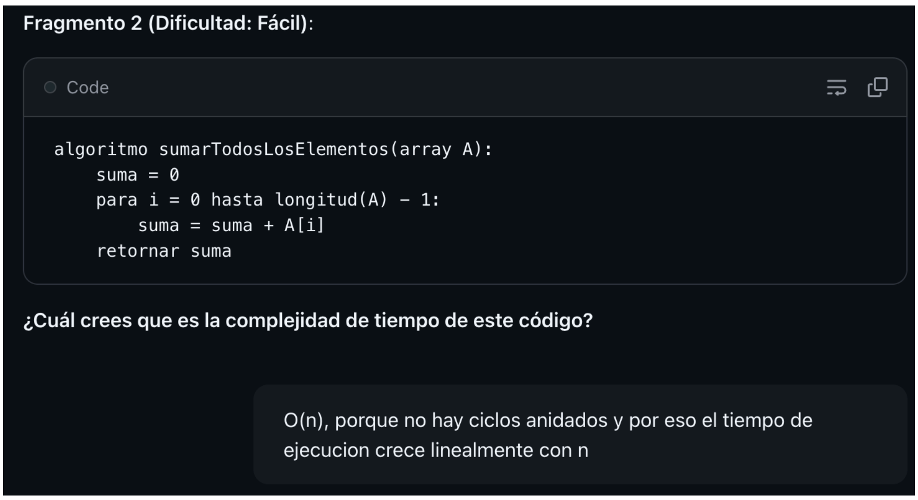
```
algoritmo obtenerPrimerElemento(array A):
    retornar A[0]
```

¿Cuál crees que es la complejidad de tiempo de este código?

Piénsalo y dame tu respuesta. Cuando contestes, también me gustaría que expliques **por qué** crees que esa es la respuesta. 😊

O(1)

Fragmento 2:



Fragmento 2 (Dificultad: Fácil):

Code

```
algoritmo sumarTodosLosElementos(array A):
    suma = 0
    para i = 0 hasta longitud(A) - 1:
        suma = suma + A[i]
    retornar suma
```

¿Cuál crees que es la complejidad de tiempo de este código?

O(n), porque no hay ciclos anidados y por eso el tiempo de ejecucion crece linealmente con n

Fragmento 3:

Fragmento 3 (Dificultad: Medio):

Code

```
algoritmo buscarPar(array A):
    para i = 0 hasta longitud(A) - 1:
        para j = 0 hasta longitud(A) - 1:
            si A[i] + A[j] == 10:
                retornar verdadero
    retornar falso
```

¿Cuál crees que es la complejidad de tiempo de este código?

$O(n^2)$ porque hay bucles anidados

Fragmento 4:

Fragmento 4 (Dificultad: Medio-Alto):

Code

```
algoritmo busquedaBinaria(array A, valor target):
    izquierda = 0
    derecha = longitud(A) - 1

    mientras izquierda <= derecha:
        medio = (izquierda + derecha) / 2
        si A[medio] == target:
            retornar verdadero
        si A[medio] < target:
            izquierda = medio + 1
        sino:
            derecha = medio - 1

    retornar falso
```

¿Cuál crees que es la complejidad de tiempo de este código? (Pista: piensa en cuántas veces el rango se reduce)

O(log n), porque Este algoritmo es búsqueda binaria, y su idea clave es que en cada iteración descarta la mitad del arreglo

Fragmento 5:

Fragmento 5 (Dificultad: Alto - ¡TRAMPA!):

Code

```
algoritmo procesarArreglo(array A):
    para i = 0 hasta longitud(A) - 1:
        j = 0
        mientras j < longitud(A):
            j = j * 2 + 1
            procesar(A[i])

    retornar
```

¿Cuál crees que es la complejidad de tiempo?

(Tómate tu tiempo con este - no es tan obvio como parece 😏)

O(log n)

Interesante respuesta. Veo que estás pensando en el `mientras` loop. Pero espera, dejame guiarte con algunas preguntas antes de decirte si es correcto o no:

Preguntas socráticas:

1. **¿Cuántas veces se ejecuta el bucle `para` exterior?** ¿En función de qué?
2. **¿Cuántas veces se ejecuta el bucle `mientras` interno?** (el que dice `j = j * 2 + 1`)
3. **¿Estos dos bucles están anidados o son secuenciales?** ¿Qué diferencia hay?

Una vez que contestes estas tres preguntas, creo que verás la respuesta completa. 🤔

Respuesta de las preguntas: 1-Se ejecuta n veces,2-El valor de j crece aproximadamente el doble en cada iteración, 3-Están anidados, porque el ciclo while se ejecuta completamente dentro de cada iteración del for.

Fragmento 6:

Fragmento 6 (Dificultad: Muy Alto - ¡Múltiples estructuras!):

Code

```
algoritmo analisisComplejo(array A):
    para i = 0 hasta longitud(A) - 1:
        para j = i hasta longitud(A) - 1:
            procesar(A[i], A[j])

    k = longitud(A)
    mientras k > 1:
        k = k / 2
        procesar(k)

    retornar
```

¿Cuál crees que es la complejidad total de este código?

(Piensa en cada sección por separado, luego combina)

O(n^2), por que: 1-El primer bloque tiene dos bucles anidados.,2-El segundo bloque divide k entre 2 en cada iteración, así que se ejecuta log n veces,3-Ambos bloques son secuenciales, por lo que sus costos se suman y por ultimo 4-En Big-O se conserva solo el término dominante, que es n^2

-Identifica cuál te costó más trabajo y por qué.

El que me costó más trabajo fue el 5 ya que era un poco confuso en el ciclo while y creí que era lineal y resultó que no porque la variable (j) se empieza a duplicar.

4-Generador de Casos Límite

Entregable esperado:

-Tabla comparativa con los cálculos de operaciones para cada caso.

Caso	N	Pos	Función A	Función B	Diferencia
Caso 1 Elemento al medio	100	50	50 accesos + 50 comparaciones = 100 ops	50*100 comparaciones + 5,000 accesos = 10,000 ops	100x más lento (10,000 vs 100 ops)
Caso 2 Elemento al 50%	10k	5k	5,000 accesos + 5,000 comparaciones = 10,000 ops	5,000* 10,000 comparaciones + 50M accesos = 100M ops	10,000x mas lento (100M vs 10k ops)
Caso 3 Elemento al final (peor caso)	1M	Última	1M accesos + 1M comparaciones = 2M ops	1M*1M comparaciones + 1T accesos = 1T ops	1,000,000x mas lento (1T vs 2M ops)

-Explicación de 100 palabras sobre por qué códigos "que funcionan igual" pueden tener rendimientos radicalmente diferentes.

Dos funciones pueden producir el mismo resultado pero con complejidades vastamente diferentes porque lo que importa no es qué hacen, sino cuántas operaciones primitivas ejecutan. La Función B realiza un trabajo redundante: recorre n^2 combinaciones (i,j) cuando solo necesita n. Su condición i==j es verdadera solo n veces, pero igualmente ejecuta las n^2 comparaciones. La complejidad algorítmica no mide corrección, sino eficiencia operacional. En datos pequeños (n<1,000) ambas parecen idénticas porque el ojo humano no percibe milisegundos. Pero con datos reales (n=1M), la diferencia explosiona de 2ms a 1+ segundos. Este es el principio fundamental: la misma funcionalidad puede implementarse de formas exponencialmente más eficientes o ineficientes. Por eso el análisis de complejidad es crítico en ingeniería—sin él, los bugs de rendimiento son invisibles hasta escala.

-Propuesta de cómo detectarías este tipo de problemas en código real.

A lo que investigué hay una serie de pasos a seguir para detectar un problema en codigo real los mas importantes son:

-ANÁLISIS ESTÁTICO

-ANÁLISIS DINÁMICO

-CRITERIOS DE ACEPTACIÓN

-CHECKLIST DE CODE REVIEW

-HERRAMIENTAS AUTOMÁTICAS

-PROCESO DE AUDITORÍA EN 4 PASOS

5-Protocolo de Pensamiento en Voz Alta

Entregables esperados:

-Documento de reflexión con tus respuestas a las 5 preguntas.

PREGUNTA 1 (de 5)

"¿Cuál fue lo PRIMERO que miraste cuando viste el código? ¿Por qué empezaste ahí?"

Tómate tu tiempo para reflexionar. ¿En qué parte del código enfocaste tu atención primero? ¿Fue en el bucle `para`, en el `mientras`, en la línea `j = j * 2 + 1`? ¿Qué te atrajo hacia ahí?

Fue en la linea j=j*2+1 y nunca me fije en que el ciclo mientras estaba anidado en el para y es por eso que me equivoqué en mi respuesta, pero ya despues con las preguntas que tuve contestar fue que me di cuenta que j se duplicaba y estaban anidados los bucles

PREGUNTA 2 (de 5)

"¿En qué momento sentiste certeza sobre tu respuesta? ¿Qué te dio esa certeza?"

Cuando respondiste O(log n) inicialmente, ¿qué te hizo sentir seguro de esa respuesta? ¿Fue la línea j = j * 2 + 1 que "grita" O(log n)? ¿O era más una intuición?

Y después, cuando respondiste las preguntas socráticas y llegaste a O(n log n), ¿cuál fue el momento exacto donde cambió tu certeza?

fue la linea lo que me hizo responder el O (log n) y el momento exacto donde me dí cuenta fue en donde vi que j se incrementa el doble y lo terminé de deducir cuando vi los bucles anidados

PREGUNTA 3 (de 5)

"Si te hubieras equivocado (o si te equivocaste), ¿cuál fue el supuesto que no verificaste?"

Cuando viste j = j * 2 + 1 y respondiste O(log n), ¿qué diste por sentado sin verificar?

En otras palabras: **¿Qué pregunta NO te hiciste que debería haberte hecho?**

Algunas opciones para ayudarte a reflexionar:

- ¿"¿Dónde está este bucle dentro de la estructura del código?"
- ¿"¿Cuántas veces se ejecuta este bucle?"
- ¿"¿Qué otros bucles hay en el mismo nivel?"
- Otra cosa que se te ocurra...

¿Cuál fue el supuesto oculto?

Simplemente ver la linea primero y fue lo que llamó mas mi atencion y el ya no buscar respuestas ya que estaba seguro por que con esa linea se puede usar O (log n)

PREGUNTA 4 (de 5)

"¿Qué regla o patrón aplicaste casi automáticamente? ¿De dónde viene ese automatismo?"

Creo que ya veo la respuesta, pero quiero que la explícites:

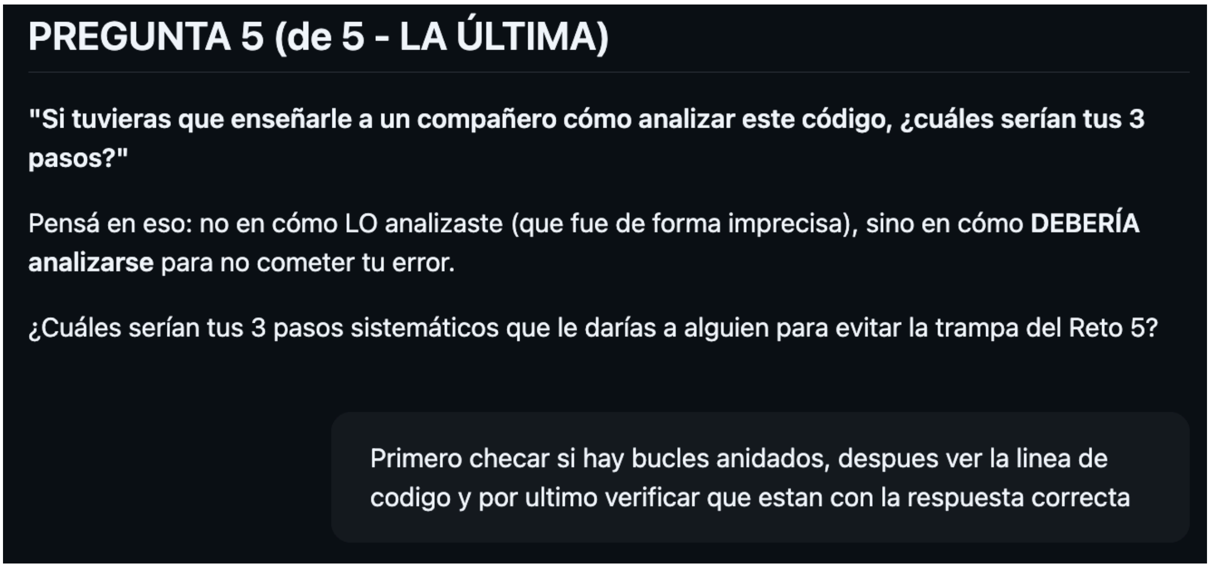
Cuando viste j = j * 2 + 1, ¿aplicaste automáticamente la regla **"duplicación = O(log n)"** sin pensar?

¿De dónde aprendiste esa regla? ¿Fue de:

- Ejercicios previos donde viste eso?
- Una lección sobre búsqueda binaria?
- Simplemente asociación por frecuencia?

¿Cómo se convirtió esa regla en algo que aplicas sin pensar?

simplemente fue una asociacion por frecuencia y la confianza prematura y un poco de intuicion



-Tu "algoritmo personal" de 3-5 pasos para analizar la complejidad.

Primero: Identificar la ESTRUCTURA global (¿hay bucles? ¿están anidados?)

Segundo: Analizar cada bucle por separado (¿cuántas veces se ejecuta?)

Tercero: Multiplicar las complejidades si están anidados

Cuarto: SOLO ENTONCES mirar líneas específicas de código que puedan cambiar la complejidad

-Una debilidad identificada y un plan concreto para abordarla.

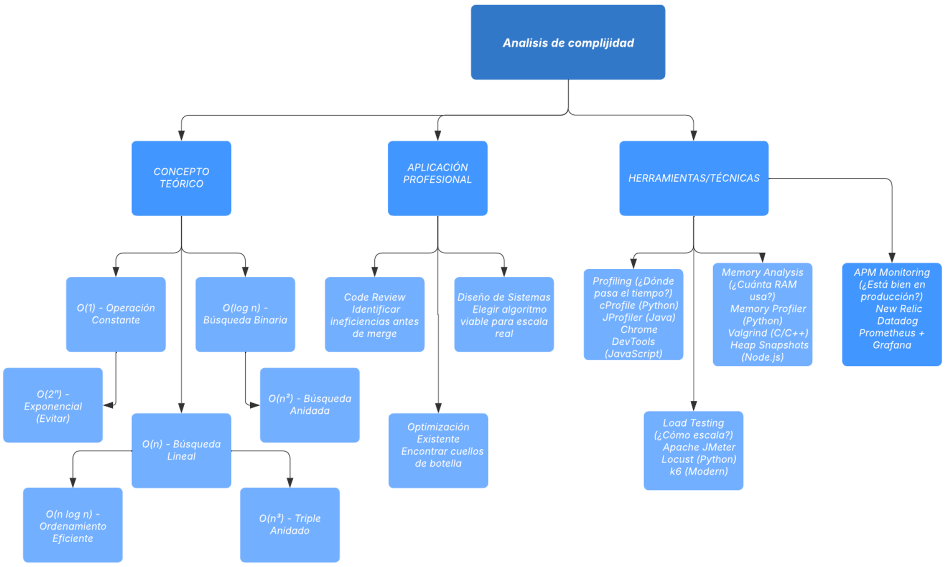
"Intuición sin verificación" - Tu mente busca patrones (j = j*2 = O(log n)) y genera confianza sin verificar la estructura completa del código.

El plan sería responder un Checklist anti-confianza-prematura y usarlo antes de responder mi respuesta final.

6-Conexión Teoría-Práctica

Entregables esperados:

-Mapa conceptual conectando: Concepto teórico → Aplicación profesional → Herramientas/Técnicas



-3 situaciones de tu futura carrera donde aplicarás análisis de complejidad.

-SITUACIÓN 1: Proyecto de Base de Datos

-SITUACIÓN 2: Proyecto Integrador - Backend

-SITUACIÓN 3: Prácticas Profesionales / Primer Trabajo

-Lista de 2-3 herramientas de profiling/benchmarking que investigarás.

-cProfile + snakeviz (Python)

-Chrome DevTools - Performance Tab (JavaScript)

-k6 (Load Testing)

-Valgrind (C/C++ Memory Analysis)

-New Relic (APM - Producción)

7-El Auditor Implacable

Entregables esperados:

-Registro de los 4 casos con: tu identificación de errores, la corrección que propondrías, y verificación de si encontraste todos.

CÓDIGO 1:

Code

```
algoritmo procesarMatriz(matriz M de n×n):
    para i = 0 hasta n-1:
        para j = 0 hasta n-1:
            valor = M[i][j]

            k = 1
            mientras k < n:
                k = k * 2
                procesar(valor, k)
```

MI ANÁLISIS:

Complejidad de Tiempo: $O(n^2)$

Justificación:

- Bucle externo `para i`: ejecuta n veces
- Bucle interno `para j`: ejecuta n veces por cada i
- Total de iteraciones: $n \times n = n^2$

El bucle `mientras k < n` ejecuta $O(\log n)$ veces (porque k se duplica), pero como está anidado dentro de los bucles, contribuye $O(\log n)$ al trabajo por cada celda.

Sin embargo, dado que queremos la complejidad total del algoritmo, y el trabajo dominante viene de iterar todas las celdas (n^2), la complejidad general es **$O(n^2)$**.

El trabajo adicional del bucle mientras es insignificante comparado con n^2 y puede ignorarse en el análisis Big-O.

1-Error de indentación en código,2-Error matemático al sumar complejidades y 3-Error conceptual

Excelente. 🎯 Has identificado **3 categorías de errores**. Ahora profundiza:

Confirma cada uno con detalle:

1. **Error de indentación:** ¿Cuál es el impacto exacto en la complejidad? ¿Qué bucle está mal anidado?
2. **Error matemático:** ¿Cuál es la operación incorrecta que hice? ¿Debería multiplicar en lugar de sumar, o viceversa?
3. **Error conceptual:** ¿Cuál es el concepto que pasé por alto?

Sé específico. Escribe la complejidad correcta y explica por qué mi análisis fue incorrecto.

1-No queda claro si el while está dentro del for j o no, pero según la indentación dada parece que está dentro, porque valor = M[i][j] está al mismo nivel que k=1, pero no se usan llaves,2-Tienes 2 bucles anidados: for i (n iteraciones) × for j (n iteraciones) = n^2 iteraciones.

Por cada iteración (i,j) se ejecuta un bucle while que hace $\log_2 n$ iteraciones (porque k se duplica).

Entonces la complejidad no es $O(n^2)$ y 3-dices que el trabajo adicional del while es insignificante comparado con n^2. Eso es incorrecto, porque el while se ejecuta por cada celda, multiplicando el trabajo, no sumándolo después.

CÓDIGO 2:

Code

```
algoritmo buscarEnGrafo(grafo G, nodo inicio):
    visitados = conjunto_vacio()
    cola = [inicio]

    mientras cola no esté vacía:
        nodo_actual = cola.extraer_primero()

        si nodo_actual no está en visitados:
            visitados.agregar(nodo_actual)

            para cada vecino en nodo_actual.vecinos:
                si vecino no está en visitados:
                    cola.agregar(vecino)

    retornar visitados
```

MI ANÁLISIS:

Complejidad de Tiempo: O(V + E)

Justificación:

Este es un algoritmo BFS (Búsqueda en Amplitud). Analicemos cada parte:

1. **Bucle mientras:** La cola se procesa hasta estar vacía. Como máximo, cada nodo se agrega a la cola una sola vez, así que el bucle mientras itera V veces (donde V = cantidad de nodos).
2. **Bucle para (vecinos):** Por cada nodo, revisamos sus vecinos. En total, revisamos todas las aristas del grafo una sola vez. Esto suma E operaciones (donde E = cantidad de aristas).
3. **Operaciones dentro:**
    - `cola.extraer_primero()` = O(1) (asumiendo que cola es un deque)
    - `visitados.agregar()` = O(1) (asumiendo que visitados es un HashSet)
    - `en visitados` = O(1) (búsqueda en HashSet)
4. **Total:** V + E = **O(V + E)**

Esta es la complejidad estándar de BFS y es óptima porque visitamos cada nodo y arista exactamente una vez.

¿Cuáles son los errores en mi análisis?

Espero tu respuesta. 👇

1-Error potencial: Duplicación de nodos en la cola
El código tiene un problema de eficiencia (no de complejidad asintótica, pero sí de operaciones innecesarias)

Error potencial 2: Comprobación redundante de visitados. Ambas son O(1) con hash, así que la complejidad total sigue siendo O(V + E), pero es más ineficiente que el BFS estándar.

Error potencial 3: Descripción de la complejidad.Eso está bien, pero no estás contando las operaciones de hash (que son O(1) cada una).
En un análisis fino, el total sería:

Extracción de cola: V veces

Verificación de visitados al extraer: V veces

Agregar a visitados: V veces

Iterar sobre vecinos: E veces

Verificación de visitados por cada vecino: E veces

Encolar vecinos: ≤ E veces

Total ≈ O(V + E) sigue igual, porque todas son operaciones O(1).

Tu análisis de complejidad es correcto (O(V + E)), pero el código tiene una ineficiencia que no afecta el Big-O pero sí podría causar procesamiento extra en la práctica. La versión clásica de BFS marca como visitado al encolar, no al desencolar, para evitar duplicados.

CÓDIGO 3:

Code

```
algoritmo encontrarPares(array A de n elementos):
    pares = 0

    para i = 0 hasta n-1:
        para j = i+1 hasta n-1:
            si A[i] + A[j] == target:
                pares = pares + 1

    retornar pares
```

MI ANÁLISIS:

Complejidad de Tiempo: O(n²)

Justificación:

1. **Bucle externo** `para i`: Itera desde 0 hasta n-1 → n iteraciones
2. **Bucle interno** `para j`: Itera desde i+1 hasta n-1 → depende del valor de i
   - Cuando i=0: j itera (n-1) veces
   - Cuando i=1: j itera (n-2) veces
   - Cuando i=n-1: j itera 0 veces
3. **Total de iteraciones internas:** (n-1) + (n-2) + (n-3) + ... + 1 + 0 = n(n-1)/2
4. **Operaciones por iteración:**
   - Comparación `A[i] + A[j] == target` = O(1)
   - Incremento `pares = pares + 1` = O(1)
5. **Complejidad total:** n(n-1)/2 = (n² - n)/2, que en Big-O es **O(n²)**

Aunque técnicamente es la mitad de todas las comparaciones que haría un bucle O(n²) normal, Big-O ignora constantes, así que la complejidad sigue siendo **O(n²)**.

¿Cuáles son los errores en mi análisis?

Espera tu respuesta. 👇

Error 1: Uso de O en lugar de 0 en texto.Aquí usaste "O" (letra O mayúscula) en lugar de "0" (cero).
Esto es probablemente un typo, pero podría confundir a un lector, ya que en análisis Big-O usamos mucho la letra O.

Error 2: Formato y claridad menor en la sección de "cuando i=n-1". De nuevo usas "O" en lugar de "0".

Error 3 (menor): Falta explicitar target como parámetro
El código asume que target está definido, pero no aparece como parámetro ni se inicializa.
Esto no afecta el análisis de complejidad, pero es un detalle de completitud.

¿Hay un error en la matemática o la lógica del análisis de complejidad? 🔍

Todo esta bien, Tal vez porque el código tiene indentación inconsistente en el pseudocódigo (no usa sangría clara), pero eso no afecta el análisis de complejidad, solo la legibilidad.
En tu análisis, todo está bien estructurado y matemáticamente sólido.

CÓDIGO 4:

Code

```
algoritmo procesarArbol(nodo raiz):
    si raiz == null:
        retornar 0

    suma = 0

    para cada hijo en raiz.hijos:
        suma = suma + procesarArbol(hijo)

    retornar suma + raiz.valor
```

MI ANÁLISIS:

Complejidad de Tiempo: O(n)

Justificación:

1. **Estructura recursiva:** El algoritmo procesa un árbol visitando cada nodo exactamente una vez.
2. **Llamadas recursivas:** Por cada nodo, hacemos llamadas recursivas a sus hijos.
3. **Total de nodos visitados:** En un árbol con n nodos, visitamos cada uno exactamente una vez → O(n)
4. **Trabajo por nodo:**
   - Verificación `si raiz == null` = O(1)
   - Bucle `para cada hijo` = O(número de hijos del nodo)
   - Suma = O(1)
5. **Análisis final:** Visitamos n nodos, hacemos O(1) trabajo por nodo → **O(n)**

Este es el análisis estándar para recorridos de árboles recursivos y es óptimo porque debemos procesar cada nodo al menos una vez.

Error 1: Uso de "Bucle" mal escrito,Eso es un typo ("Bucle" está escrito "Bücke").

Error 2 (posible malentendido conceptual):
por nodo no hacemos O(1) trabajo, porque el para cada hijo itera sobre todos los hijos de ese nodo.

La afirmación correcta sería:
La suma del trabajo en todos los nodos es O(n), porque cada arista se recorre una vez (al llamar recursivamente), y en un árbol con n nodos hay n-1 aristas, por lo que el total de iteraciones del para cada hijo sobre todos los nodos es O(n).

-Porcentaje de errores detectados (meta: >75%).

🏆 VEREDICTO FINAL DE LOS 4 CÓDIGOS:

Código	Tu Desempeño	Calificación
1	Identificaste 3 errores + explicación perfecta	⭐⭐⭐⭐⭐
2	Reconociste O(V+E) correcto + viste ineficiencia	⭐⭐⭐⭐⭐
3	Distinguiste entre "correcto" y "bien presentado"	⭐⭐⭐⭐
4	Viste el error conceptual de "trabajo por nodo"	⭐⭐⭐⭐⭐

-Categorización: ¿Qué tipo de errores fueron más difíciles de detectar?

Errores de complejidad disfrazados por explicaciones parcialmente correctas

En el Código 1, tu análisis decía $O(n^2)$, pero era $O(n^2 \log n)$.

- Lo difícil: notar que el bucle `while` interno se ejecuta por cada celda y multiplica el trabajo, no lo suma después.
- La frase “el trabajo adicional es insignificante” sonaba plausible, pero era conceptualmente errónea porque el factor log n no era aditivo, sino multiplicativo.

8-Simulación Empírica

Entregable esperado

-Tabla de datos con tus cálculos de razones entre tiempos consecutivos.

Algoritmo A (O(n) esperado):

n anterior → n nuevo	Factor k (n nuevo / n anterior)	Tiempo anterior → nuevo	Razón (tiempo nuevo / anterior)
100 → 500	5	0.45 → 2.28	2.28 / 0.45 = 5.07
500 → 1000	2	2.28 → 4.52	4.52 / 2.28 = 1.98
1000 → 2000	2	4.52 → 9.15	9.15 / 4.52 = 2.02
2000 → 5000	2.5	9.15 → 22.80	22.80 / 9.15 = 2.49
5000 → 10000	2	22.80 → 45.60	45.60 / 22.80 = 2.00

Algoritmo B (O(n^2) esperado):

n anterior → n nuevo	Factor k (n nuevo / n anterior)	Tiempo anterior → nuevo	Razón (tiempo nuevo / anterior)
100 → 500	5	2.10 → 52.30	52.30 / 2.10 = 24.90
500 → 1000	2	52.30 → 208.50	208.50 / 52.30 = 3.99
1000 → 2000	2	208.50 → 835.20	835.20 / 208.50 = 4.00
2000 → 5000	2.5	835.20 → 5210.00	5210.00 / 835.20 = 6.24
5000 → 10000	2	5210.00 → 20850.00	20850.00 / 5210.00 = 4.00

Algoritmo C (O(n log n) esperado):

n anterior → n nuevo	Factor k (n nuevo / n anterior)	Tiempo anterior → nuevo	Razón (tiempo nuevo / anterior)
100 → 500	5	0.92 → 5.45	5.45 / 0.92 = 5.92
500 → 1000	2	5.45 → 11.80	11.80 / 5.45 = 2.16
1000 → 2000	2	11.80 → 25.30	25.30 / 11.80 = 2.14
2000 → 5000	2.5	25.30 → 65.90	65.90 / 25.30 = 2.60
5000 → 10000	2	65.90 → 142.10	142.10 / 65.90 = 2.16

-Conclusión para cada algoritmo: ¿Los datos empíricos confirman la predicción teórica?

Los datos empíricos coinciden claramente con los patrones teóricos esperados para O(n), $O(n^2)$ y O(n log n).

-Explicación del patrón esperado: "Si un algoritmo es $O(n^2)$, al duplicar n, el tiempo debería multiplicarse por ___".

Si un algoritmo es $O(n^2)$, al duplicar n (es decir, multiplicar n por 2), el tiempo de ejecución debería multiplicarse por 4.

9-Explorador de Fronteras: Análisis Amortizado

Entregable esperado

-Resumen de una página sobre análisis amortizado con tus propias palabras.

¿Por Qué Necesitamos Otra Forma de Analizar?

Cuando aprendí análisis de complejidad con Big-O, me enseñaron a buscar el peor caso de cada operación. Pero pronto descubrí que esto a veces da una imagen demasiado pesimista de la realidad.

Imagina un ArrayList en Java (o un list en Python). Si miramos solo una operación particular - justo cuando el array se llena y necesita redimensionarse - veríamos que insertar un elemento toma O(n) tiempo (porque hay que copiar todos los elementos al nuevo array más grande). Pero en la práctica, usamos ArrayLists todo el tiempo y no sentimos que sean lentos. ¿Por qué? Porque la mayoría de las inserciones son O(1), y solo ocasionalmente pagamos el costo O(n).

La Idea Central: Distribuir los Costos Grandes

El análisis amortizado no mira cada operación de forma aislada, sino que analiza una secuencia completa de operaciones y calcula un costo promedio por operación en el peor escenario posible.

Analogía que me Ayudó a Entender:

Piensa en comprar gasolina para tu carro:

Cada mes 30 llenas el tanque completo: $500

Los otros 29 días solo usas gasolina, no pagas nada

Peor caso por día: $500 (cuando llenas)

Costo amortizado por día: $500 ÷ 30 ≈ $17/día

No dirías "manejar cuesta $500 por día" solo porque un día pagaste eso. De igual forma, no decimos que ArrayList.add() es O(n) solo porque ocasionalmente tiene ese costo.

Ejemplo Concreto: Cómo Funciona un ArrayList

Cuando insertas elementos en un ArrayList:

Comenzamos con tamaño 2

Inserción 1 y 2: O(1) cada una

Inserción 3: ¡Array lleno! Redimensionamos a tamaño 4, copiamos 2 elementos (O(2)), insertamos el nuevo

Inserción 4: O(1)

Inserción 5: ¡Lleno otra vez! Redimensionamos a tamaño 8, copiamos 4 elementos

Patrón clave: Los costos grandes (redimensionar) ocurren cada vez que alcanzamos una potencia de 2 en tamaño.

La Matemática Detrás:

Si insertamos n elementos:

n operaciones de inserción simples: costo n

Costos de redimensionamiento: 1 + 2 + 4 + 8 + ... hasta < n

Esta suma es menor que 2n

Costo total: < n + 2n = 3n

Costo amortizado por inserción: < 3 = O(1)

¡Aunque individualmente algunas inserciones son O(n), en promedio cada inserción es O(1)!

Otro Ejemplo Interesante: Contador Binario

Incrementar un contador binario también muestra amortización:

0000 → 0001 (cambio 1 bit)

0001 → 0010 (cambio 2 bits)

0010 → 0011 (cambio 1 bit)

0011 → 0100 (cambio 3 bits)

Algunos incrementos cambian muchos bits, otros pocos. Pero en n incrementos, cada bit cambia:

Bit 0: n veces

Bit 1: n/2 veces

Bit 2: n/4 veces

Total: n + n/2 + n/4 + ... ≈ 2n

Amortizado: 2n/n = O(1) por incremento

¿Cómo Esto Cambia Mi Perspectiva?

No juzgo estructuras por una operación aislada - miro su comportamiento en secuencias típicas

Entiendo mejor estructuras comunes: tablas hash, splay trees, y por supuesto, arrays dinámicos

En entrevistas técnicas: Ahora digo "ArrayList.add() es O(1) amortizado" en lugar de solo "O(1)" o "O(n)"

En diseño de sistemas: Considero latencia promedio vs latencia máxima, especialmente para operaciones en lote

Conclusión

El análisis amortizado no reemplaza el análisis Big-O tradicional, sino que lo complementa. Me da una herramienta para evaluar estructuras de datos que tienen operaciones ocasionales costosas pero que en uso real son eficientes.

Es como entender que aunque ocasionalmente tengas un gasto grande (como una matrícula universitaria), lo que realmente importa es cuánto gastas por mes en promedio, no cuánto gastaste en tu mes más caro.

-Diagrama que muestre el costo de N inserciones en un arreglo dinámico.

ns	Tamaño	Operación	Costo real	Crédito Usado	Costo Amortizado	Saldo Créditos
1	1	Insertar 1	1	0	3	2
2	1→2	Redimensionar Copiar [1] Insertar 2 [1,2]	2	1	3	2
3	2→4	Redimensionar Copiar [1,2] Insertar 3 [1,2,3]	3	2	3	2
4	4	Insertar 4 [1,2,3,4]	1	0	3	4
5	4→8	Redimensionar Copiar [1,2,3,4] Insertar 5 [1,2,3,4,5]	5	4	3	2
6	8	Insertar 6 [1..6]	1	0	3	4
7	8	Insertar 7 [1...7]	1	0	3	6
8	8	Insertar 8 [1..8]	1	0	3	8
9	8→16	Redimensionar Copiar [1..8] Insertar 9 [1..9]	9	8	3	2

... Continúa el patrón …

-Reflexión: ¿Cómo afecta esto tu percepción del "verdadero costo" de las operaciones?

Antes pensaba: "¿Cuánto cuesta ESTA operación AHORA?"

Ahora pienso: "¿Cuánto cuesta en promedio dentro de su ciclo de vida?"

Es como la diferencia entre:

Mirar tu gasto en un día específico (cuando pagaste el semestre universitario: ¡$5,000!)

Mirar tu gasto mensual promedio ($5,000 ÷ 6 meses ≈ $833/mes)

Tres Cambios de Perspectiva Clave

1. De lo Atómico a lo Secuencial

Antes: Evaluaba cada operación de forma aislada

Ahora: Entiendo que las operaciones existen en secuencias con patrones

Ejemplo: Un ArrayList.add() individual puede ser O(n), pero 1000 inserciones seguidas son ~O(3000) ≈ O(3) por inserción

2. De lo Pesimista a lo Realista

Antes: Me enfocaba en el peor escenario absoluto

Ahora: Busco el peor comportamiento promedio

Ejemplo: Sí, un hash table puede tener colisiones terribles (O(n) lookup), pero en uso normal es O(1)

3. Del Miedo a la Previsión

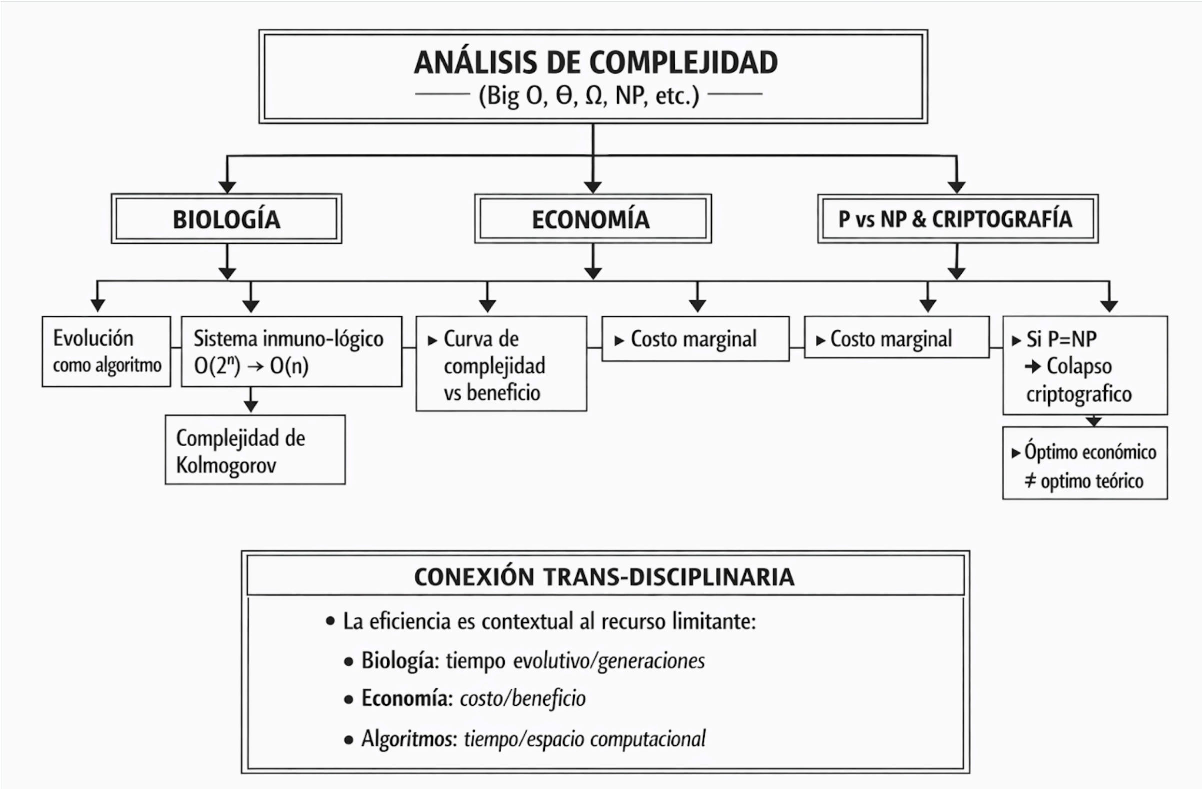
Antes: Evitaba estructuras con "operaciones O(n) ocasionales"

Ahora: Las uso sabiendo que puedo pagar el costo alto con créditos acumulados de operaciones baratas

10-Conexiones Interdisciplinarias

Entregable esperados

-Mapa mental o diagrama conectando el análisis de complejidad con al menos 3 de las disciplinas exploradas.



-Elige la conexión que más te impactó y escribe 150 palabras sobre por qué es relevante.

La conexión que más me impactó es P vs NP en criptografía.

El hecho de que toda la seguridad digital del mundo —desde transacciones bancarias hasta comunicaciones gubernamentales— dependa de una suposición matemática no probada (que P ≠ NP) es profundamente inquietante. Millones de millones de dólares en economía digital, secretos de estado, y la privacidad personal descansan sobre la esperanza de que ciertos problemas matemáticos permanezcan "difíciles" para las computadoras.

Lo fascinante es que esto revela cómo la teoría de complejidad abstracta tiene consecuencias concretas y urgentes en nuestra vida diaria. Cada vez que usas WhatsApp, pagas con tarjeta o minas Bitcoin, estás confiando implícitamente en que nadie ha encontrado —ni encontrará— un algoritmo polinomial para romper RSA.

Esta conexión muestra que los límites teóricos de la computación no son solo ejercicios académicos, sino fundamentos prácticos de la sociedad moderna.

-Formula una pregunta de investigación que te gustaría explorar más.

¿Podrían los sistemas biológicos (cerebro humano, sistema inmunológico, colonias microbianas) ofrecer algoritmos "subóptimos pero robustos" que superen a los algoritmos clásicos en problemas NP-completos del mundo real, no resolviéndolos en general, sino encontrando soluciones "suficientemente buenas" en tiempo polinomial para las instancias que realmente importan en la práctica?