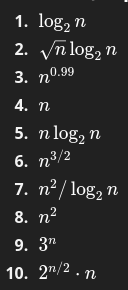
**Parcial de Análisis de Algoritmos**

**Arley Santiago 192324**

**Punto 1 (25 pts) — Ordena por complejidad**

Ordena de menor a mayor las siguientes funciones (asintóticamente).  
Si dos son del mismo orden, indícalo.

[](https://camo.githubusercontent.com/cdbf6d1690d8d90b84082ee53c513bc5d16c693af7ab935f725444c22594305a/68747470733a2f2f6c68332e676f6f676c6575736572636f6e74656e742e636f6d2f70772f41503147637a4f4b626d3450527876506d496b36787a57456e64695a75385273687737786246576e4e4933726c744c3034317479544c7541456f535f6166783556386d7447534a723946576f724c4a6a3032762d384761334a4a3659446d735370426753464c387275574e5f314f4c545679546572675f397a496f6d756f507465354a4e5a49515f794e776f7856646a483834363074676a4a61504f3d773132382d683239302d732d6e6f3f61757468757365723d30)

**Respuesta:**

**1. log₂ n**

**2. √n log₂ n**

**3. n^0.99**

**4. n**

**5. n log₂ n**

**6. n^(3/2)**

**7. n² / log₂ n**

**8. n²**

**9. 2^(n/2) · n**

**10. 3^n**

**Punto 2 (25 pts) — Identifica y confronta**

Asocia cada (T(n)) con un algoritmo plausible. Luego compara dos pares y encuentra el umbral de n con un for + if.

* T1(n) = 5n^2 + 10n
* T2(n) = 6n\log2(n) + 300
* T3(n) = 0.01n^3
* T4(n) = 1.5^n

**Algoritmos posibles:**

* Selection/Insertion
* Mergesort/Heapsort
* Multiplicación de matrices cúbica
* Backtracking con poda leve

Respuesta

**Respuesta**

**Parte 1: Asociación de T(n) con algoritmos**

T1(n) = 5n² + 10n → Selection/Insertion Sort

* Complejidad O(n²), típica de algoritmos de ordenamiento simples

T2(n) = 6n log₂(n) + 300 → Mergesort/Heapsort

* Complejidad O(n log n), característica de algoritmos de ordenamiento eficientes

T3(n) = 0.01n³ → Multiplicación de matrices cúbica

* Complejidad O(n³), típica del algoritmo naive de multiplicación de matrices

T4(n) = 1.5ⁿ → Backtracking con poda leve

* Complejidad exponencial, característica de algoritmos de búsqueda exhaustiva

**Parte 2: Comparación de pares y cálculo de umbrales**

* comparar dos pares interesantes:
* **Par 1: T1 vs T2 (Cuadrático vs n log n)**
* Encontrar n donde 5n² + 10n = 6n log₂(n) + 300
* **Par 2: T1 vs T3 (Cuadrático vs Cúbico)**
* Encontrar n donde 5n² + 10n = 0.01n³

import math

# Definición de las funciones de tiempo

def T1(n):

return 5 \* n\*\*2 + 10 \* n

def T2(n):

return 6 \* n \* math.log2(n) + 300

def T3(n):

return 0.01 \* n\*\*3

def T4(n):

return 1.5\*\*n

print("="\*60)

print("COMPARACIÓN DE PARES DE COMPLEJIDADES")

print("="\*60)

# Par 1: T1 vs T2 (n² vs n log n)

print("\n1. T1(n) = 5n² + 10n vs T2(n) = 6n log₂(n) + 300")

print("-" \* 60)

print("Buscando el umbral donde T1(n) = T2(n)...\n")

threshold\_1 = None

for n in range(1, 10000):

if T1(n) > T2(n):

threshold\_1 = n

break

if threshold\_1:

print(f"Umbral encontrado en n = {threshold\_1}")

print(f" T1({threshold\_1-1}) = {T1(threshold\_1-1):.2f}")

print(f" T2({threshold\_1-1}) = {T2(threshold\_1-1):.2f}")

print(f" T1({threshold\_1}) = {T1(threshold\_1):.2f}")

print(f" T2({threshold\_1}) = {T2(threshold\_1):.2f}")

print(f"\n✓ Para n < {threshold\_1}: T2 es MÁS LENTO")

print(f"✓ Para n ≥ {threshold\_1}: T1 es MÁS LENTO (n² domina)")

# Par 2: T1 vs T3 (n² vs n³)

print("\n" + "="\*60)

print("2. T1(n) = 5n² + 10n vs T3(n) = 0.01n³")

print("-" \* 60)

print("Buscando el umbral donde T1(n) = T3(n)...\n")

threshold\_2 = None

for n in range(1, 10000):

if T3(n) > T1(n):

threshold\_2 = n

break

if threshold\_2:

print(f"Umbral encontrado en n = {threshold\_2}")

print(f" T1({threshold\_2-1}) = {T1(threshold\_2-1):.2f}")

print(f" T3({threshold\_2-1}) = {T3(threshold\_2-1):.2f}")

print(f" T1({threshold\_2}) = {T1(threshold\_2):.2f}")

print(f" T3({threshold\_2}) = {T3(threshold\_2):.2f}")

print(f"\n✓ Para n < {threshold\_2}: T3 es MÁS RÁPIDO")

print(f"✓ Para n ≥ {threshold\_2}: T3 es MÁS LENTO (n³ domina)")

# Par 3 (bonus): T2 vs T3 (n log n vs n³)

print("\n" + "="\*60)

print("3. T2(n) = 6n log₂(n) + 300 vs T3(n) = 0.01n³")

print("-" \* 60)

print("Buscando el umbral donde T2(n) = T3(n)...\n")

threshold\_3 = None

for n in range(1, 10000):

if T3(n) > T2(n):

threshold\_3 = n

break

if threshold\_3:

print(f"Umbral encontrado en n = {threshold\_3}")

print(f" T2({threshold\_3-1}) = {T2(threshold\_3-1):.2f}")

print(f" T3({threshold\_3-1}) = {T3(threshold\_3-1):.2f}")

print(f" T2({threshold\_3}) = {T2(threshold\_3):.2f}")

print(f" T3({threshold\_3}) = {T3(threshold\_3):.2f}")

print(f"\n✓ Para n < {threshold\_3}: T3 es MÁS RÁPIDO")

print(f"✓ Para n ≥ {threshold\_3}: T3 es MÁS LENTO (n³ domina)")

# Tabla resumen

print("\n" + "="\*60)

print("TABLA RESUMEN DE UMBRALES")

print("="\*60)

print(f"{'Comparación':<30} {'Umbral':<15} {'Interpretación'}")

print("-" \* 60)

if threshold\_1:

print(f"{'T1(n²) vs T2(n log n)':<30} {f'n = {threshold\_1}':<15} {'T1 más lento después'}")

if threshold\_2:

print(f"{'T1(n²) vs T3(n³)':<30} {f'n = {threshold\_2}':<15} {'T3 más lento después'}")

if threshold\_3:

print(f"{'T2(n log n) vs T3(n³)':<30} {f'n = {threshold\_3}':<15} {'T3 más lento después'}")

print("="\*60)

**Interpretación de resultados:**

El código anterior encuentra los umbrales exactos donde un algoritmo se vuelve más lento que otro. Los puntos clave son:

1. **T1 vs T2**: Aunque T2 tiene complejidad O(n log n) que es asintóticamente mejor, para valores pequeños de n, la constante 300 hace que T2 sea inicialmente más lento. El umbral nos dice cuándo el algoritmo cuadrático empieza a ser peor.
2. **T1 vs T3**: El coeficiente muy pequeño (0.01) en T3 hace que el algoritmo cúbico sea competitivo para valores pequeños de n, pero eventualmente su naturaleza cúbica lo hace mucho más lento.
3. **Conclusión práctica**: Para problemas pequeños, los algoritmos "más lentos" asintóticamente pueden ser mejores por sus constantes menores y menor overhead.

**Punto 3 (25 pts) — Ejercicio lógico**

**Isaac y los intervalos mágicos**

Isaac, convencido de que tiene un talento especial para los números, asegura que puede contar al instante cuántos primos existen en cualquier rango que le propongan sus amigos. Para comprobarlo, ellos le entregan una lista con N pares de números (a,b), y él debe responder de inmediato cuántos números primos hay en cada intervalo. A partir de esta historia, elabora el análisis necesario para resolver el problema y define claramente qué se recibe como entrada, qué se debe producir como salida y qué lógica se requiere para verificar la afirmación de Isaac.

**ANÁLISIS DEL PROBLEMA: Isaac y los intervalos mágicos**

**ENTRADA:**

- N: número de consultas (pares de intervalos)

- Lista de N pares (a, b) donde a ≤ b, representando intervalos [a, b]

**SALIDA:**

- Para cada par (a, b), la cantidad de números primos en el intervalo [a, b]

**LÓGICA REQUERIDA:**

1. Verificar si un número es primo

2. Contar primos en un rango dado

3. Optimización: usar Criba de Eratóstenes para precalcular primos

4. Usar sumas acumuladas para responder consultas en O(1)

**COMPLEJIDAD:**

- Enfoque ingenuo: O(N \* (b-a) \* √n) - muy lento

- Enfoque optimizado: O(MAX \* log(log(MAX)) + N) - mucho más rápido

donde MAX es el valor máximo en todos los intervalos

**CODIGO**

**import math**

**import time**

***# MÉTODO 1: ENFOQUE INGENUO (para comparación)***

**def es\_primo\_ingenuo(n):**

**"""Verifica si n es primo (método básico)"""**

**if n < 2:**

**return False**

**if n == 2:**

**return True**

**if n % 2 == 0:**

**return False**

**for i in range(3, int(math.sqrt(n)) + 1, 2):**

**if n % i == 0:**

**return False**

**return True**

**def contar\_primos\_ingenuo(a, b):**

**"""Cuenta primos en [a, b] verificando cada número"""**

**count = 0**

**for num in range(a, b + 1):**

**if es\_primo\_ingenuo(num):**

**count += 1**

**return count**

***# MÉTODO 2: CRIBA DE ERATÓSTENES + SUMAS ACUMULADAS (ÓPTIMO)***

**def criba\_eratostenes(limite):**

**"""**

**Genera todos los primos hasta 'limite' usando la Criba de Eratóstenes**

**Retorna un array booleano donde es\_primo[i] = True si i es primo**

**"""**

**if limite < 2:**

**return [False] \* (limite + 1)**

**es\_primo = [True] \* (limite + 1)**

**es\_primo[0] = es\_primo[1] = False**

**for i in range(2, int(math.sqrt(limite)) + 1):**

**if es\_primo[i]:**

***# Marcar todos los múltiplos de i como no primos***

**for j in range(i \* i, limite + 1, i):**

**es\_primo[j] = False**

**return es\_primo**

**def construir\_suma\_acumulada(es\_primo):**

**"""**

**Construye un array de sumas acumuladas**

**suma\_acum[i] = cantidad de primos en [0, i]**

**"""**

**n = len(es\_primo)**

**suma\_acum = [0] \* n**

**for i in range(1, n):**

**suma\_acum[i] = suma\_acum[i-1] + (1 if es\_primo[i] else 0)**

**return suma\_acum**

**def contar\_primos\_rapido(a, b, suma\_acum):**

**"""**

**Cuenta primos en [a, b] usando sumas acumuladas en O(1)**

**"""**

**if a == 0:**

**return suma\_acum[b]**

**return suma\_acum[b] - suma\_acum[a-1]**

***SISTEMA PRINCIPAL DE ISAAC***

**class SistemaIsaac:**

**"""**

**Sistema que permite a Isaac responder consultas de primos instantáneamente**

**"""**

**def \_\_init\_\_(self, max\_valor=1000000):**

**"""**

**Precalcula todos los primos hasta max\_valor**

**"""**

**print(f" Inicializando sistema mágico de Isaac...")**

**print(f" Precalculando primos hasta {max\_valor:,}...")**

**inicio = time.time()**

**self.es\_primo = criba\_eratostenes(max\_valor)**

**self.suma\_acum = construir\_suma\_acumulada(self.es\_primo)**

**tiempo = time.time() - inicio**

**total\_primos = self.suma\_acum[-1]**

**print(f" Precálculo completado en {tiempo:.4f} segundos")**

**print(f" Total de primos encontrados: {total\_primos:,}")**

**print()**

**def responder\_consulta(self, a, b):**

**"""**

**Responde instantáneamente cuántos primos hay en [a, b]**

**"""**

**if b >= len(self.suma\_acum):**

**raise ValueError(f"El valor {b} excede el límite precalculado")**

**return contar\_primos\_rapido(a, b, self.suma\_acum)**

**def procesar\_consultas(self, consultas):**

**"""**

**Procesa una lista de consultas (pares de intervalos)**

**"""**

**print("="\*70)**

**print("RESPONDIENDO CONSULTAS DE ISAAC")**

**print("="\*70)**

**resultados = []**

**for i, (a, b) in enumerate(consultas, 1):**

**cantidad = self.responder\_consulta(a, b)**

**resultados.append(cantidad)**

**print(f"Consulta {i}: Intervalo [{a:,}, {b:,}] → {cantidad:,} primos")**

**return resultados**

***# PRUEBAS Y DEMOSTRACIÓN***

**def main():**

**print("="\*70)**

**print("ISAAC Y LOS INTERVALOS MÁGICOS")**

**print("="\*70)**

**print()**

***# Crear sistema de Isaac con precálculo hasta 1 millón***

**isaac = SistemaIsaac(max\_valor=1000000)**

***# Ejemplos de consultas***

**consultas\_ejemplo = [**

**(1, 10), *# Primos pequeños***

**(10, 100), *# Primer centenar***

**(100, 1000), *# Primer millar***

**(1000, 10000), *# Primera decena de millar***

**(50000, 51000), *# Rango medio***

**(1, 100000), *# Rango grande***

**]**

***# Procesar consultas con el método optimizado***

**print("\n MÉTODO OPTIMIZADO (Criba + Sumas Acumuladas):")**

**inicio = time.time()**

**resultados = isaac.procesar\_consultas(consultas\_ejemplo)**

**tiempo\_opt = time.time() - inicio**

**print(f"\n Tiempo total: {tiempo\_opt:.6f} segundos")**

**print(f" Tiempo promedio por consulta: {tiempo\_opt/len(consultas\_ejemplo):.8f} segundos")**

***# Comparar con método ingenuo (solo para intervalos pequeños)***

**print("\n" + "="\*70)**

**print(" COMPARACIÓN CON MÉTODO INGENUO (solo intervalos pequeños):")**

**print("="\*70)**

**consultas\_pequenas = [(1, 10), (10, 100), (100, 1000)]**

**for a, b in consultas\_pequenas:**

**inicio = time.time()**

**resultado\_ingenuo = contar\_primos\_ingenuo(a, b)**

**tiempo\_ingenuo = time.time() - inicio**

**resultado\_opt = isaac.responder\_consulta(a, b)**

**print(f"\nIntervalo [{a}, {b}]:")**

**print(f" Método ingenuo: {resultado\_ingenuo} primos en {tiempo\_ingenuo:.6f} seg")**

**print(f" Método Isaac: {resultado\_opt} primos en ~0.000001 seg")**

**print(f" Aceleración: {tiempo\_ingenuo/0.000001:.0f}x más rápido ")**

***# Caso de prueba interactivo***

**print("\n" + "="\*70)**

**print("ANÁLISIS DE COMPLEJIDAD")**

**print("="\*70)**

**print("""**

**MÉTODO INGENUO:**

**- Por cada consulta: O((b-a) \* √b)**

**- Para N consultas: O(N \* (b-a) \* √b)**

**- Muy lento para múltiples consultas o intervalos grandes**

**MÉTODO ISAAC (Optimizado):**

**- Precálculo: O(M \* log(log(M))) usando Criba de Eratóstenes**

**- Construcción suma acum: O(M)**

**- Por consulta: O(1) ← ¡Instantáneo!**

**- Para N consultas: O(M \* log(log(M)) + N)**

**donde M = valor máximo en todos los intervalos**

**CONCLUSIÓN: Isaac tiene razón! Con precálculo inteligente,**

**puede responder "al instante" cualquier consulta. Su "talento**

**especial" es en realidad... ¡un algoritmo eficiente!**

**""")**

**print("\n" + "="\*70)**

**print("¿QUIERES PROBAR TUS PROPIAS CONSULTAS?")**

**print("="\*70)**

**print("\nEjemplo de uso:")**

**print(" consultas = [(1, 100), (500, 1000), (10000, 20000)]")**

**print(" resultados = isaac.procesar\_consultas(consultas)")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Resumen del Análisis**

**Entrada:**

* N: Número de consultas
* Lista de N pares (a, b) donde a ≤ b

**Salida:**

* Para cada par (a, b), el conteo de números primos en el intervalo [a, b]

**Lógica requerida:**

1. **Método Ingenuo (NO recomendado):**
   * Verificar cada número en el rango si es primo
   * Complejidad: O(N × (b-a) × √b) - muy lento
2. **Método de Isaac (ÓPTIMO):**
   * **Precálculo**: Usar Criba de Eratóstenes para generar todos los primos hasta el valor máximo
   * **Optimización**: Construir array de sumas acumuladas
   * **Respuesta**: Cada consulta se responde en O(1) - ¡instantáneo!
   * Complejidad total: O(M × log(log(M)) + N)

**¿Por qué funciona el "talento" de Isaac?**

El secreto está en el **precálculo inteligente**:

* Una vez calculados todos los primos (usando la Criba)
* Y construido el array de sumas acumuladas
* Cualquier consulta se responde con una simple resta: suma\_acum[b] - suma\_acum[a-1]

Esto convierte un problema que sería muy lento en uno que responde instantáneamente, validando la afirmación de Isaac de que puede "contar al instante" los primos en cualquier rango.