ANALISIS DE ALGORITMOS TPI P1

COMISION 10- BATISTA FEDERICO BOVIO JOSE

ANALISIS BIG-O

Introducción Teórica a la Notación Big O

La notación Big O es una herramienta fundamental en el análisis de algoritmos, ya que nos permite describir el comportamiento asintótico de una función. Es decir, nos ayuda a entender cómo crece el tiempo de ejecución (o el uso de memoria) de un algoritmo a medida que el tamaño de la entrada (n) tiende a infinito.

¿Qué representa Big O?

- Cuando decimos que un algoritmo es **O(f(n))**, estamos definiendo un **límite superior** en su crecimiento.
- Esta notación nos permite comparar la eficiencia de distintos algoritmos **independientemente del hardware o lenguaje de programación**.
- Además, Big O se enfoca en el **peor caso**, aunque también existen notaciones para el mejor caso y el caso promedio.

Tipos comunes de complejidades

Notación Big O	Nombre	Ejemplo
O(1)	Constante	Acceso a un elemento en un array
O(log n)	Logarítmica	Búsqueda binaria
O(n)	Lineal	Recorrer una lista
O(n log n)	Lineal logarítmica	Algoritmos de ordenación eficientes (Merge Sort)
$O(n^2)$	Cuadrática	Ordenamiento burbuja, selección
$O(2^n)$	Exponencial	Problemas de fuerza bruta

Aplicación de Big O en los Algoritmos de Números Pares

Análisis de tiempo en pares_con_if(n)

- El bucle for itera n veces.
- Las operaciones dentro del bucle (if y append) tienen complejidad O(1).

- Por lo tanto, el tiempo total es **O(n)** (complejidad lineal).

Análisis de espacio

- La lista pares almacena aproximadamente n/2 elementos → **O(n)** de espacio.

Análisis de pares_con_range(n)

Tiempo:

- La llamada a range(2, n+1, 2) se genera de forma perezosa → **O(1)**.
- Convertirla a lista (list(...)) toma O(n/2) ≈ **O(n)**.
- Total: **O(n)** (aunque con una constante más pequeña que en pares_con_if).

Espacio:

- Se almacenan n/2 elementos → **O(n)**.

Resultados Obtenidos y Comparación

Tabla Comparativa (Big O)

Algoritmo	**Tiempo (Big O)**	**Espacio (Big O)**	**Observaciones**
pares_con_if	O(n)	O(n)	Más lento por el bucle e if.
pares_con_range	O(n)	O(n)	Más rápido gracias a optimizaciones internas.

¿Por qué pares_con_range es más rápido si ambos son O(n)?

Aunque ambos algoritmos tienen complejidad **lineal**, pares_con_range es más eficiente porque:

- Evita un bucle explícito en Python, que suele ser más lento que las operaciones internas en C.
- No realiza verificaciones condicionales (if i % 2 == 0) en cada iteración.
- La función range() está altamente optimizada en Python (está implementada en C).

Gráfica de Tiempos Empíricos

En nuestras pruebas observamos lo siguiente:

- Ambas curvas crecen linealmente (O(n)) cuando graficamos en escala logarítmica.
- pares_con_range presenta una pendiente menor, lo que indica una constante oculta más baja.
- La diferencia se vuelve más notable cuando usamos valores grandes de n (por ejemplo, n = 10^7).

Conclusión

- Ambos algoritmos tienen la **misma complejidad teórica (O(n))**.
- En la práctica, pares_con_range es más eficiente gracias a:
    - Menor sobrecarga del intérprete de Python (al apoyarse en operaciones en C).
    - La eliminación de operaciones redundantes como el if.

Recomendamos usar pares_con_range para este problema.

1. Algoritmo pares_con_if(n)

```python
def pares_con_if(n):
    pares = []
    for i in range(1, n + 1):
        if i % 2 == 0:
            pares.append(i)
    return pares
```

- pares = []: Crea una lista vacía → **O(1)**.
- for i in range(1, n + 1): Itera n veces → **O(n)**.
- if i % 2 == 0: Verifica paridad → **O(1)** por iteración.
- pares.append(i): Añade elementos a la lista → O(1) amortizado, ocurre en n/2 iteraciones.
- return pares: Devuelve la lista → **O(1)**.

Resumen de Complejidad:

- **Tiempo total**: O(n) × O(1) = **O(n)**.
- **Espacio**: La lista almacena n/2 elementos → **O(n)**.

2. Algoritmo pares_con_range(n)

```python
def pares_con_range(n):
    return list(range(2, n + 1, 2))
```

- range(2, n + 1, 2): Define un generador → O(1).
- list(...): Convierte el rango en lista → O(n/2) ≈ O(n).
  Resumen de Complejidad:
- Tiempo: O(n).
- Espacio: O(n) (n/2 elementos almacenados).

Comparación Detallada

Factor	**pares_con_if (Bucle + If)**	**pares_con_range (Range con Paso)**
¿Usa bucle en Python?	Sí (más lento)	No (usa range, optimizado en C)
Operaciones por iteración	2 (if + append)	1 (generación directa)
Acceso a memoria	Varias asignaciones	Asignación contigua eficiente
Overhead de Python	Alto	Bajo (implementación en C)

Resultados Empíricos Esperados

- Para n = 10,000,000:
  - pares_con_if: ~0.5 segundos (según hardware).
  - pares_con_range: ~0.05 segundos (10 veces más rápido).
- La diferencia crece a medida que aumentamos n.

Gráfica Teórica vs. Práctica

- Teóricamente, ambos algoritmos son **O(n)** → líneas rectas en escala logarítmica.

- En la práctica, pares_con_range tiene una pendiente menor, lo que refleja una **mejor eficiencia real**.

Conclusión Final

- La notación Big O nos indica que ambos algoritmos son O(n), pero la **implementación importa**.
- pares_con_range es preferible porque:
    1. Evita un bucle explícito en Python.
    2. No hace verificaciones if innecesarias.
    3. Utiliza range(), que está optimizado en C.

Cuando trabajamos con valores grandes de n, **conviene usar pares_con_range**.

¿Cuándo usamos Big O en la vida real?

- Cuando necesitamos elegir entre algoritmos al trabajar con grandes volúmenes de datos.
- Para optimizar código en sistemas críticos como motores de búsqueda o bases de datos.
- En entrevistas técnicas: empresas como Google, Meta y Amazon suelen evaluar comprensión de complejidad algorítmica.

Este análisis nos muestra que, aunque Big O **no lo es todo**, sí es una herramienta esencial para escribir código más eficiente y tomar decisiones técnicas informadas.