Carlos Javier Bravo Intriago
17 de dic. de 2025 23:40

Hola a todos,

Siguiendo el debate interesante que tuvimos en la primera clase sobre el ejercicio de listas, el profesor me encomendó profundizar un poco más en por qué ciertas soluciones presentan un comportamiento cuadrático y otras lineal.

Preparé este notebook donde comparo el rendimiento de la solución con `list comprehension`, la variante del bucle `for` y la de `set`. Los resultados confirman el impacto de los bucles anidados en listas grandes, pero también destacan que las soluciones basadas en iterar listas eran más amigables con la memoria RAM.

Les dejo aquí los resultados y también incluí una solución extra que intenta buscar un equilibrio. Me encantaría conocer sus opiniones, correcciones, feedback o si ven algún detalle que se me haya pasado por alto.

https://colab.research.google.com/drive/1TSyWUitOgS7Cf8TBK5rnsyKRYjkyYslx (https://colab.research.google.com/drive/1TSyWUitOgS7Cf8TBK5rnsyKRYjkyYslx)

Como dato extra, ya que en la clase de hoy tocamos algo sobre el algoritmo de Heron, la ecuación iterativa para encontrar la raiz cuadrada se puede conseguir también con el método de newton-rapson, en el siguiente enlace propongo una version donde encuentro un valor inicial o un `magic number` para que en una o dos iteraciones se pueda encontrar un resultado muy aproximado con un error absoluto cercano a 10^{-5} .

https://leetcode.com/problems/sqrtx/solutions/6071226/fast-sqrt-in-python/ (https://leetcode.com/problems/sqrtx/solutions/6071226/fast-sqrt-in-python/)
https://aliennerd.dev/sqrt (https://aliennerd.dev/sqrt)

¡Felices Fiestas!

Carlos Javier Bravo Intriago
12 de ene. de 2026 18:13 (Editado por Carlos Javier Bravo Intriago el 14 de ene. de 2026 12:07)

Respondo este post para aclarar las preguntas antes sugeridas en clases, como también para hacer una especie de Fe de erratas en uno de los puntos donde calculaba la complejidad computacional de una de las variantes.

Fe de Erratas:

En este apartado cabe aclarar que no he considerado la conversión o el `cast` del `set` `result` a una `list`. En mi versión previa he omitido esa conversión que al final es de complejidad lineal $O(R)$.

Volviendo a recalcular la complejidad (no el número de operaciones) de esta variante, la conversión o `cast` de `list` a `set` es de complejidad lineal, las cuales representé como:

- `set(list_a)` : $O(N)$
- `set(list_b)` : $O(M)$

Mi afirmación de que el método `set.intersection()` itera sobre el conjunto de menor cardinalidad y verifica pertenencia o existencia en el conjunto de mayor cardinalidad, no es algo meramente especulativo. Hago esta afirmación en base al código fuente de CPython (https://github.com/python/cpython/blob/main/Objects/setobject.c#L1649-L1653) donde internamente el método `set_intersection()` (entre 1649 y 1653) hace un `swap` o un intercambio de variables para cumplir lo que he mencionado previamente.

```c
static PyObject *
set_intersection(PySetObject *so, PyObject *other)
{
    PySetObject *result;
    PyObject *key, *it, *tmp;
    Py_hash_t hash;
    int rv;

    if ((PyObject *)so == other)
        return set_copy_impl(so);

    result = (PySetObject *)make_new_set_basetype(Py_TYPE(so), NULL);
    if (result == NULL)
        return NULL;

    if (PyAnySet_Check(other)) {
        Py_ssize_t pos = 0;
        setentry *entry;

        if (PySet_GET_SIZE(other) > PySet_GET_SIZE(so)) {
            tmp = (PyObject *)so;
            so = (PySetObject *)other;
            other = tmp;
        }
```

Aquí he interpretado que la complejidad de encontrar la intersección de 2 conjuntos es de complejidad $O(min(N, M))$ haciendo referencia a que es la complejidad en función del conjunto con menor cardinalidad.

Podemos reinterpretar en Python esta afirmación de la forma (he omitido validaciones extra que están en el código fuente original en C):

```python
lowest_cardinality_set = set(list_a)
highest_cardinality_set = set(list_b)
intersection_result = set()
# swap
if len(highest_cardinality_set) < len(lowest_cardinality_set):
    highest_cardinality_set, lowest_cardinality_set = lowest_cardinality_set, highest_cardinality_set

for item in lowest_cardinality_set:  # O(min(M, N)) complexity
    if item in highest_cardinality_set:  # O(1) complexity
        intersection_result.add(item)

intersection_result
```

En la explicación de la variante 4, mencioné que el uso de la palabra reservada `in` en el contexto de diccionarios o conjuntos (`set`), en el caso más típico de uso o en el caso donde no existan colisiones, la búsqueda se hace en tiempo constante $O(1)$.

Por otra parte, si se usa la palabra reservada `in` en el contexto de `listas` o `tuplas`, esta invoca internamente al método `__contains__()`, el cual es prácticamente un bucle `for` para buscar elemento a elemento. Ref (https://stackoverflow.com/a/55032431)

Calculando la complejidad final para la variante 1, sería la adición de sus complejidades de sus subrutinas por separado, quedando:

$$O(N) + O(M) + O(min(M, N)) + O(R)$$

Dado que el término $O(min(N, M))$ está acotado superiormente por $O(N)$ y $O(M)$ (es decir, $min(M,N) \leq N+M$), este término es absorbido por la suma de las variables dominantes. Por ende, la expresión se reduce a:

$$O(N+M+R)$$

Esto indica una complejidad lineal, donde el tiempo de ejecución crece linealmente en proporción a la suma de los tamaños de las entradas N, M y R.

¿Realmente he conseguido una complejidad O(N) con hashmap?

En el escenario donde no existan colisiones, puedo afirmar que he conseguido una complejidad lineal $O(N)$.

¿En todos los casos?

Esta pregunta tiene matices y trataré de aclararlos acotando cierto aspecto.

Primera acotación, estamos tratando con enteros de 32 o de 64 bits, no otros tipos como flotantes u otros objetos hashables.
El hash de un entero en Python es el propio entero, siempre que se mantenga dentro del rango de un entero nativo de máquina.

```python
hash(5) == 5
hash(42) == 42
hash(-7) == -7
```

Sin embargo, internamente se aplica una reducción modular para evitar desbordamientos en arquitecturas de 32 o 64 bits.

```
hash(x) = x % _PyHASH_MODULUS
```

Donde _PyHASH_MODULUS (https://docs.python.org/3.2/library/stdtypes.html#hashing-of-numeric-types) depende de la plataforma (normalmente 2^{61-1} o 2^{31-1}).

Esto garantiza que el resultado sea un entero dentro del rango de hash válido inclusive para números extremadamente grandes dentro del rango de `_PyHASH_MODULUS`.

Por ende, con enteros, una forma de generar colisiones en un `hashmap` o `hastable` es cuando los números superan a `_PyHASH_MODULUS`. Como he propuesto el código de ejemplo, no hemos superado el rango de 2^{61-1} , por lo que la probabilidad de existir colisiones es prácticamente nula.

Otra forma curiosa de generar una colisión es cuando el valor es `-1` y `-2`. El motivo es algo curioso y se puede encontrar en el siguiente enlace (https://docs.python.org/3/library/stdtypes.html#hashing-of-numeric-types).

¿Y qué es un hash?

Según lo describen Sedgewick y Wayne en Algorithms (4.ª ed.), una tabla de hash (hash map) representa un método fundamental para la implementación de tablas de símbolos, cuyo principio rector consiste en transformar una clave de búsqueda directamente en una dirección de memoria dentro de un arreglo, permitiendo así el acceso inmediato a los valores asociados. El motor matemático detrás de esta estructura es la función hash, un algoritmo determinista encargado de convertir cualquier tipo de clave —como una cadena de texto o un objeto complejo— en un índice entero dentro del rango [0, M-1], donde M corresponde al tamaño de la tabla. Para que el sistema sea eficiente, los autores enfatizan que una función hash robusta debe satisfacer la "suposición de hashing uniforme" (uniform hashing assumption), distribuyendo las claves de manera equitativa para minimizar las colisiones y garantizar operaciones de búsqueda e inserción en tiempo constante amortizado $O(1)$, gestionando así el clásico compromiso entre eficiencia temporal y consumo de espacio (Sedgewick & Wayne, 2011, pp. 458-462).

¿Y si no tenemos en nuestro lenguaje algo como hash

Si no existe en el lenguaje de programación la función hash y los hashmaps, este ejercicio no podría resolverse con una complejidad temporal $O(N)$, sino que se requerirían estrategias de ordenamiento y búsqueda para optimizar la complejidad y evitar caer en el escenario de $O(N^2)$.

¿Y si ordenamos previamente las listas ?

Consideremos el código ejemplo:

```
list_a.sort()
result = []

for b_item in list_b:
    for a_item in list_a: # sorted list
        result.append(a_item)
```

Ordenar una lista en el mejor escenario implicaría una complejidad cuasi-lineal $O(N\log N)$. Si solo nos quedásemos con el ordenamiento y aplicáramos las estrategias que no impliquen hashmaps previamente definidos en el notebook (es decir, las variantes de complejidad $O(N^2)$, la complejidad final quedaría de la forma:

$$O(N\log N) + O(M \times N)$$

El término cuadrático *absorbe* al término cuasi-lineal, quedando una complejidad temporal de:

$$O(M \times N)$$

Esto sugiere que la complejidad es del orden cuadrático.

Ahora consideremos el código ejemplo aplicando una estrategia extra:

```
# inspirado en el codigo ejemplo de https://docs.python.org/3/library/bisect.html#searching-sorted-lists
import bisect #  implementación optimizada de búsqueda binaria en Python

lowest_cardinality_list = list_a
highest_cardinality_list = list_b

if len(highest_cardinality_list) < len(lowest_cardinality_list):
    lowest_cardinality_list, highest_cardinality_list = highest_cardinality_list, lowest_cardinality_list

lowest_cardinality_list.sort()
result = []

for b_item in highest_cardinality_list:
    idx = bisect.bisect_left(lowest_cardinality_list, b_item)
    if idx < len(lowest_cardinality_list) and lowest_cardinality_list[idx] == b_item:
        result.append(b_item)
```

En esta variante, si a la estrategia de ordenar la lista con menor cardinalidad ($O(N\log N)$), y se suma una búsqueda binaria, o algún otro algoritmo óptimo de búsqueda, podríamos llegar a una complejidad computacional cuasi-lineal.

En este punto, **he solicitado a Gemini** que deconstruya el código sugerido para obtener el cálculo de la complejidad temporal:

Fase 1: Pre-procesamiento (Preparación)

- ***Acción:*** *Ordenar la lista con menor cardinalidad (N).*
- ***Justificación:*** *Se elige la lista pequeña para minimizar el costo del ordenamiento y reducir la base logarítmica de las búsquedas futuras.*

- *Complejidad de la Fase:*

$$O(N\log N)$$

Fase 2: Procesamiento (Bucle e Intersección)

- *Esta fase combina tus dos puntos restantes en un flujo lógico:*

1. ***Iteración (El Bucle):***
    - *Se ejecuta un bucle for iterando sobre cada elemento de la lista con mayor cardinalidad (M).*

- *Esto define que realizaremos el proceso interno M veces.*

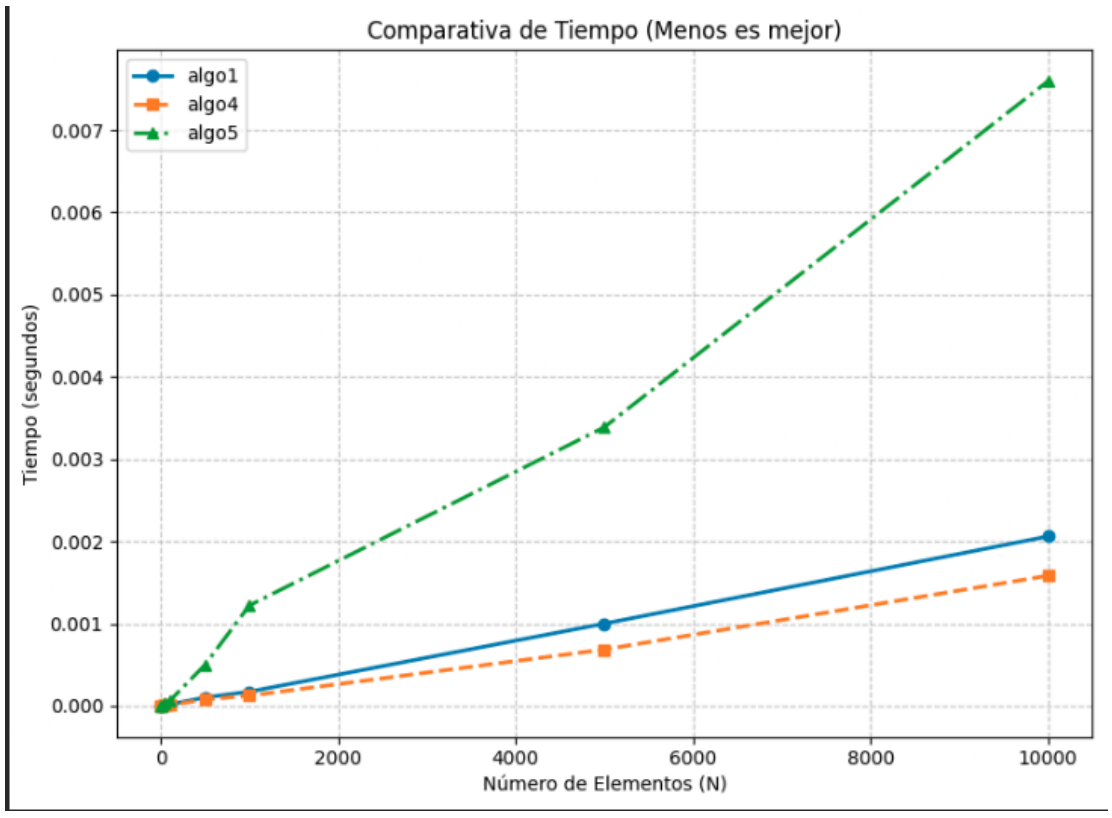
1. ***Búsqueda (El Núcleo):***
    - *Para cada elemento iterado, se ejecuta una **Búsqueda Binaria** sobre la lista ordenada (N).*
    - *Costo unitario (por elemento):* $O(\log N)$.
    - *Costo acumulado (todo el bucle):* $M \times O(\log N)$.
- *Complejidad de la Fase:*

$$O(M \log N)$$

Aterrizando este punto, la complejidad temporal usando estas estrategias en el caso promedio sería cuasi-lineal:

$$O((N+M)\log N)$$

Adjunto gráfica de las variantes lineales vs. la variante cuasi-lineal.



Referencias:

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

https://docs.python.org/3/c-api/hash.html#c.PyHASH_MODULUS (https://docs.python.org/3/c-api/hash.html#c.PyHASH_MODULUS)

https://github.com/python/cpython/blob/main/Include/cpython/pyhash.h#L18-L20 (https://github.com/python/cpython/blob/main/Include/cpython/pyhash.h#L18-L20)

https://docs.python.org/3/library/stdtypes.html#set.intersection (https://docs.python.org/3/library/stdtypes.html#set.intersection)

https://docs.python.org/3.2/library/stdtypes.html#hashing-of-numeric-types (https://docs.python.org/3.2/library/stdtypes.html#hashing-of-numeric-types)

https://github.com/python/cpython/blob/main/Objects/setobject.c#L1649-L1653 (https://github.com/python/cpython/blob/main/Objects/setobject.c#L1649-L1653)

https://docs.python.org/3/library/stdtypes.html#numeric-hash (https://docs.python.org/3/library/stdtypes.html#numeric-hash)

https://docs.python.org/3/library/bisect.html#searching-sorted-lists (https://docs.python.org/3/library/bisect.html#searching-sorted-lists)

https://stackoverflow.com/a/55032431 (https://stackoverflow.com/a/55032431)

Créditos:

Algoritmo Variante 1 usando set: Carlos Javier Bravo

Algoritmo Variante 2 usando list-comprehension: Mark William Sánchez Palacios

Algoritmo Variante 3 Bucle for anidado: Alejandro Bascuñana Giner

Algoritmo Variante 4 Híbrido entre set-comprehension y listas: Carlos Javier Bravo

Algoritmo Variante 5 Uso de estrategia basada en ordenamiento y búsqueda binaria: Raúl Reyero (Sugerencia de Ordenamiento) y Carlos Bravo (Búsqueda Binaria)

Cálculo de complejidad computacional para algoritmo con estrategia de ordenamiento y búsqueda binaria: Gemini