Análisis de Algoritmos en Python

### Trabajo Integrador - Programación I

**Índice**

1. Introducción
2. Marco Teórico

1. Caso Práctico
2. Metodología Utilizada
3. Resultados Obtenidos
4. Conclusiones
5. Bibliografía

1. Anexos

**Alumnos:**

Benjamin Ataide – benjaminatai@gmail.com

Natanael Caliva – [calivanatanaelmdz@gmail.com](mailto:calivanatanaelmdz@gmail.com)

**Materia:** Programación I

**Profeso**r: Cinthia Rigoni

**Fecha de Entrega:** 9 de junio de 2025

## **Introducción**

A veces, cuando escribimos código, nos concentramos en que simplemente funcione. Si devuelve el resultado correcto, parece suficiente. Pero a medida que los programas tienen que manejar más datos o funcionar en situaciones más exigentes, empezamos a notar que no todos los algoritmos rinden igual. Algunos tardan demasiado o consumen mucha memoria, y eso puede convertirse en un problema real.

En este trabajo quisimos explorar justamente eso: cómo elegir algoritmos que no solo resuelvan un problema, sino que lo hagan de forma eficiente. Para eso, tomamos un ejemplo simple pero muy común en la vida real: encontrar el producto más vendido dentro de una lista de ventas.

Implementamos dos soluciones distintas en Python: una más básica y otra optimizada. Después, analizamos su comportamiento tanto desde la teoría (usando la notación Big-O) como en la práctica, midiendo el tiempo real que tarda cada una.

Más allá del resultado técnico, lo que buscamos fue entender mejor por qué algunos algoritmos son más eficientes que otros y cómo aplicar ese conocimiento en proyectos reales. Esta experiencia nos ayudó a ver que analizar algoritmos no es solo una cuestión de teoría, sino una herramienta fundamental para cualquier persona que programe.

.

## **Marco Teórico**

### **• ¿Qué es un algoritmo?**

Un algoritmo es una secuencia finita y ordenada de pasos que permiten resolver un problema específico.

La **eficiencia de un algoritmo** se puede estudiar desde dos aspectos principales:

* **Eficiencia temporal**: se refiere al tiempo que tarda en ejecutarse. A mayor cantidad de datos, mayor puede ser el impacto.
* **Eficiencia espacial**: indica cuánta memoria extra necesita el algoritmo mientras se ejecuta.

### **• Análisis Teórico**

Permite estimar su rendimiento sin necesidad de probarlo con datos reales.

Este enfoque consiste en contar la cantidad de operaciones que realiza el algoritmo en función del tamaño de los datos de entrada, y expresar ese comportamiento usando la **notación Big-O**. Esta notación nos dice cómo crece el tiempo de ejecución (o el uso de memoria) a medida que crecen los datos.

* **O(1)** – **Constante**: siempre tarda lo mismo, no importa cuántos datos haya.
* **O(n)** – **Lineal**: el tiempo crece proporcionalmente a la cantidad de elementos.
* **O(n²)** – **Cuadrático**: el tiempo se dispara muy rápido, especialmente con grandes volúmenes.

### **• Análisis Empírico**

Se basa en medir el tiempo real de ejecución del algoritmo con datos concretos. Para esto se suelen usar herramientas como el módulo **time** de Python, que permite saber cuánto tarda un bloque de código desde que empieza hasta que termina.

## **Caso Práctico: Producto Más Vendido**

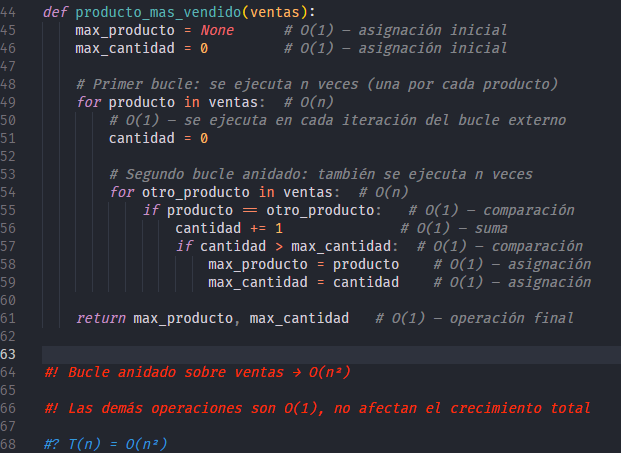
### **Problema:**

En sistemas de ventas, comercio electrónico o gestión de stock, es común querer saber cuál fue el producto más vendido en determinado periodo. Este tipo de consulta permite tomar decisiones importantes, como reponer inventario o hacer promociones.

El desafío consiste en procesar una lista de ventas donde cada elemento representa un identificador de producto, y determinar cuál de ellos aparece más veces. Aunque esto puede resolverse con distintos algoritmos, no todos son igual de eficientes. En este trabajo analizamos dos enfoques: uno simple pero lento, y otro más optimizado.

### **Algoritmo 1: Comparación Forzada (Ineficiente)**

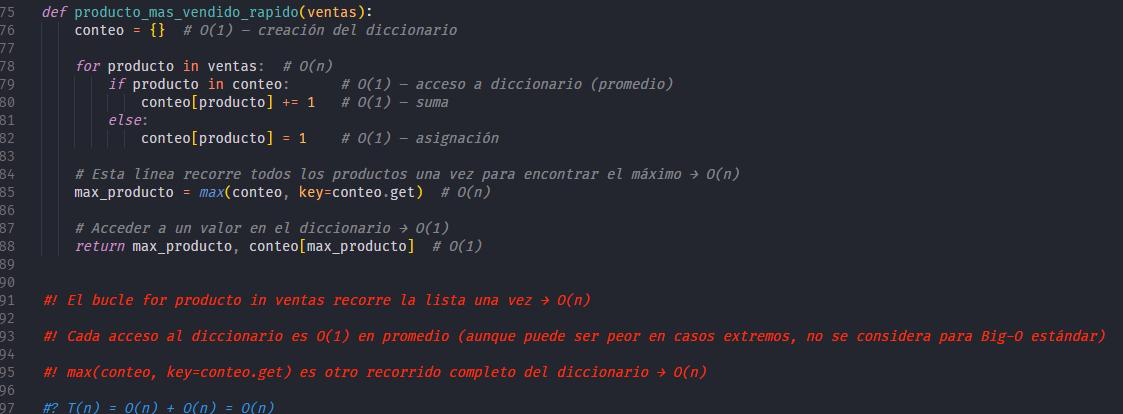
Este algoritmo compara cada producto con todos los demas usando dos bucles anidados. Debido a esto, su complejidad temporal es cuadrática **O(n²)**, lo que significa que el numero total de operaciones crece muy rápido a medida que aumenta el tamaño de la lista. Por ejemplo: si hay 10.000 productos, el algoritmo puede hacer hasta 100 millones de comparaciones.



**Complejidad: T(n) = O(n²)**

### **Algoritmo 2: Recuento con Diccionario (Eficiente)**

Este algoritmo recorre la lista de ventas una sola vez para contar las apariciones de cada producto, lo que lo hace muy eficiente. Luego usa la función max() para encontrar el producto más frecuente. En total realiza dos pasadas sobre los datos ( O(n) + O (n) ), por lo que su complejidad temporal es O(n). Esta eficiencia lo convierte en una excelente opción para manejar grandes volúmenes de información.



**Complejidad: T(n) = O(n)**

## 

## **Metodología Utilizada**

1. Para hacer este trabajo, empezamos por entender bien qué queríamos resolver: encontrar cuál fue el producto más vendido a partir de una lista de ventas. Una vez planteado el problema, se nos ocurrió comparar dos formas distintas de resolverlo: una más básica y directa, y otra más eficiente.
2. Primero, analizamos cómo se comporta cada algoritmo en teoría, usando la notación Big-O para estimar cuántas operaciones hacen dependiendo del tamaño de los datos. Después, los programamos en Python, tratando de mantener el código claro y fácil de probar.
3. Para probarlos con datos reales, generamos listas de entre 1.000 y 10.000 ventas aleatorias usando el módulo random. Luego, medimos el tiempo de ejecución de cada algoritmo con el módulo time, para ver cuánto tardaba cada uno y compararlos de forma más concreta
4. Nos organizamos dividiendo las tareas: uno se enfocó en el algoritmo lento, el otro en el rápido, y después nos pusimos de acuerdo para analizar los resultados juntos. Así, cada uno pudo entender bien su parte y también ver las diferencias entre ambos enfoques. Nos sirvió mucho para aprender cómo cambia el rendimiento de un programa según cómo esté pensado.

## **Resultados Obtenidos**

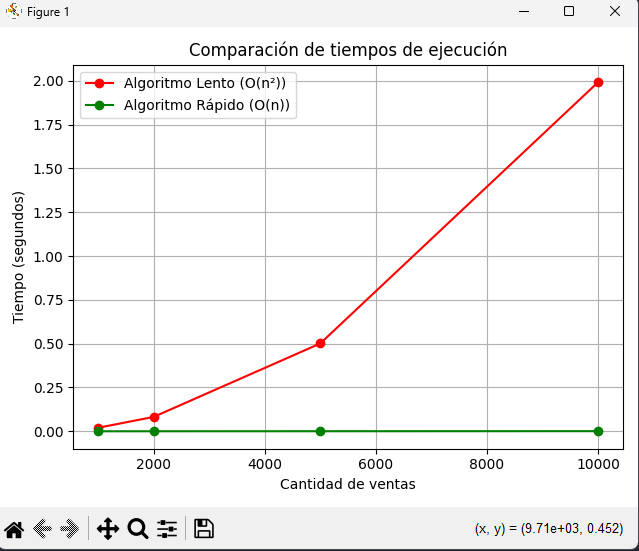
Después de implementar los dos algoritmos, realizamos pruebas para medir cuánto tardaban en ejecutarse con listas de distintos tamaños: 1.000, 2.000, 5.000 y 10.000 elementos. Para generar los datos usamos el módulo random, y para calcular los tiempos de ejecución, el módulo time.

Los resultados fueron los siguientes:

* **Algoritmo lento (O(n²))**: Tiempo estimado con 10.000 elementos: ~2.00 segundos
* **Algoritmo optimizado (O(n))**: Tiempo estimado con 10.000 elementos: ~0.00 segundos

Ambos algoritmos devolvieron el mismo resultado correcto, pero la diferencia en rendimiento fue notable. Esto demuestra que el análisis teórico (Big-O) se confirma también en la práctica.

Para visualizar esta diferencia, generamos un gráfico que muestra cómo el algoritmo lento va creciendo en tiempo de forma acelerada, mientras que el rápido se mantiene casi constante.



## **Conclusiones**

* Hacer este trabajo nos sirvió mucho para entender que no basta con que el código “funcione”. A veces dos algoritmos hacen exactamente lo mismo, pero uno tarda segundos y el otro casi ni se nota. Al principio eso puede parecer algo técnico o teórico, pero cuando lo probamos con datos reales y vimos las diferencias de tiempo, fue mucho más claro.
* Analizar la eficiencia usando Big-O y después comprobarlo con tiempos reales fue una forma muy buena de ver cómo la teoría se aplica en la práctica. El gráfico y las pruebas nos ayudaron a visualizar por qué es importante pensar bien cómo resolver un problema, sobre todo si se trata de grandes volúmenes de datos.
* Además, trabajar en equipo nos permitió dividir tareas, aprender de los dos enfoques y después unir todo para entenderlo mejor. Cada uno aportó algo distinto y eso hizo que el resultado final sea más completo
* En resumen, este trabajo no solo nos enseñó a comparar algoritmos, sino a pensar más como programadores: no solo resolver el problema, sino hacerlo de forma eficiente y consciente

## **Bibliografía**

Videos proporcionados por la cátedra.

1. Video 1: Introducción al análisis de algoritmos.
2. Video 2: Análisis empírico de sumN.
3. Video 3: Análisis empírico de sumGaus.
4. Video 1: Introducción al análisis teórico de algoritmos.
5. Video 2: Análisis teórico de las estructuras de control.
6. Video 3: Análisis teórico de sumN y sumGaus.
7. Video 4: Primer ejemplo de análisis teórico.
8. Video 5: Segundo ejemplo análisis teórico.
9. Video 1: Introducción a BIG-O.
10. Video 2: Tipos de órdenes de BIG-O.
11. Video 3: Ejemplo de cómo obtener la BIG-O.

Libreria para ver en graficos: https://matplotlib.org/stable/api/pyplot\_summary.html

## 

## **Anexos**

* Cuadro Comparativo

| **Aspectos** | **Algoritmo 1** | **Algoritmo 2** |
| --- | --- | --- |
| **Big-O** | O(n²) | O(n) |
| **Velocidad** | Lento (11 s con 100k) | Rápido (0.02 s con 100k) |
| **Memoria** | Baja (O(1)) | Media (O(n)) |
| **Código** | Muy simple | Más eficiente |
| **Escalabilidad** | Mala | Excelente |

* Captura del código fuente completo (archivo .py)

